6

Performance Prediction and Tuning of Parallel Programs

Mark Edward Crovella

Technical Report 573
August 1994

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Performance Prediction and Tuning of Parallel Programs

by

Mark Edward Crovella

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Thomas J. LeBlanc

Department of Computer Science
College of Arts and Science

University of Rochester
Rochester, New York

1994

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1994	technical report

4. TITLE AND SUBTITLE

Performance Prediction and Tuning of Parallel Programs

5. FUNDING NUMBERS

ONR N00014-92-J-1801 / ARPA 8930

6. AUTHOR(S)

Mark E. Crovella

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research
Information Systems
Arlington VA 22217

ARPA
3701 N. Fairfax Drive
Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 573

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

parallel performance measurement; analysis; prediction; lost cycles

15. NUMBER OF PAGES

115 pages

16. PRICE CODE

free to sponsors; else $5.00

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

Curriculum Vitae

Mark Edward Crovella was born in Buffalo, New York on October 28, 1959. He grew up in Amherst, New York, attending Nardin Academy and Amherst Central High School. He entered Cornell University in 1978, studying in the Department of Natural Resources. In 1982 he built a computer simulation of Red-Winged Blackbird damage to corn crops, and earned a Bachelor of Science degree from the College of Agriculture and Life Sciences. Computing soon became his main interest, and from 1982 to 1984 he worked as a Programmer for the Management Information Systems Branch of the Colorado Department of Highways, in Denver. Returning to Buffalo in 1984, he was employed at Calspan Corporation, eventually as a Senior Computer Scientist. After receiving an M.S. in Computer Science from the State University of New York at Buffalo, he entered the graduate program in Computer Science at the University of Rochester in the fall of 1989. During his first two years at Rochester, he participated in the design and implementation of the Psyche operating system, and served as a teaching assistant for courses in Parallel Programming and Introductory Computing. In 1992 he was awarded a DARPA Fellowship in Parallel Processing, followed in 1993 by an ARPA Fellowship in High Performance Computing. During his last three years at Rochester he served as a research assistant for Professor Thomas LeBlanc working on performance measurement and prediction of parallel programs.

Acknowledgments

Research in Rochester's Computer Science Department has been a wonderful experience, and the lion's share of the credit goes to Tom LeBlanc, my advisor. I want to thank him both for his work in guiding this thesis, and in shaping my graduate career. He has consistently kept his eye on the big picture and has always been ready to talk about this work. I particularly want to thank him for identifying the importance of performance measurement and prediction as a research topic, and seeing the resulting research through its various ups and downs.

Systems research at Rochester was especially enjoyable for me because of the department's atmosphere of collaboration. I want to thank Michael Scott for many helpful discussions and comments, and for the support and encouragement he's provided during my time at Rochester. I also want to thank the other members of my committee, Wei Li and William Richmond, for their time and for their comments on this work.

Much of the fun in doing research at Rochester has come from the informal groups that have formed around common interests, from operating systems, through scheduling and performance prediction, to architecture. I've learned a lot from my fellow students as we've worked together, and I want to thank them for providing demanding standards combined with constructive attitudes. In particular, I want to thank Evangelos Markatos, Cezary Dubnicki, Ricardo Bianchini, Leonidas Kontothassis, and Bob Wisniewski for the many enjoyable hours of discussion, and for the considerable joint work to which they contributed. In addition I wish to thank Lawrence Crowl for his work in our study of subgraph isomorphism; he wrote the program we used and performed many of the experiments.

Bart Miller has provided many helpful comments on this research, and served as an informal member of my committee, for which I am grateful. I want to thank Donna Bergmark for providing an excellent research environment at the Cornell Theory Center, for her helpful comments throughout this research, and for being the first user of the performance tools developed in this work. I also wish to thank Wagner Meira for being an early and patient user of tools under development.

I've had the benefit of many great colleagues at Calspan Corporation before and during my graduate studies, and I want to thank them for a stimulating and

lively work environment. In particular I want to thank Tom Jasinski and Ed Camardo for providing me with challenging opportunities, for creating a supportive atmosphere, and above all, for their friendship.

A number of friends made the bumpy road of graduate life more bearable. George Ferguson and Marc Light have kept my head above water in Rochester. Mark and Debbie Chadsey always had their door open for Linda and I. Thanks for helping us to stay sane! Mary and Bob Oberther have provided friendship and support without limit; thank you Mary for your loving care of Emily these five years.

My brothers, Paul and Robert, have been a constant inspiration to me spiritually and intellectually. They've reminded me by example of what can be accomplished, and been good friends along the way. My parents, Edward and Margaret Crovella, instilled in me their strong faith in God, and taught me the joy of learning. Coupled with my Dad's advice to "find work you enjoy doing," their actions set me on the path that led to this point. Paul and Robert, Mom and Dad, thank you.

My children Benjamin and Emily have reminded me in a million ways what matters in life. I'll always be grateful for their help in remembering that soccer and dance, alphabets and arithmetic, and walks in the park are just as important as experiments and lectures and papers.

Finally, no part of this work, nor any part of my graduate career, would have been possible without the loving support of my wife Linda. For all your patience and effort in keeping our household and lives intact, and for all the times I got home long after the kids were asleep, Linda, thank you.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930), and by a ARPA Research Assistantship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

Abstract

Parallel programs often behave in unexpected ways due to the complex relationship between the structure of a parallel program, the machine on which it is run, the number of processors used, the program's input, and the measured running time of the program. As a result, performance tuning of parallel programs is an error-prone, time-consuming process.

This dissertation describes a set of tools and methods for assisting the programmer in finding the best-performing implementation for a parallel program, and in answering common questions that arise during the performance tuning process. Our approach is based on three contributions: 1) new metrics for the measurement of parallel applications; 2) a new approach to the analysis of parallel program performance; and 3) a new modelling method that allows the programmer to predict the performance of a program in advance of a complete implementation. The metrics, which we call performance predicates, provide measurements that are amenable to analysis, and yet completely capture parallel overheads. The analysis method, lost cycles analysis, applies algorithmic analysis to parallel overheads, assisted by an on-line tool. The modelling method allows lost cycles analysis to be applied to program fragments, and provides rules for aggregating analytic results into a model for the execution time of a (possibly not-yet-implemented) parallel application. We use implementations of subgraph isomorphism and 2D FFT on the SGI Challenge Series and KSR1 multiprocessors to illustrate our methods and tools, and show how our approach can be used to explain surprising performance results and predict the performance of alternative implementations of an application in advance of implementation, while avoiding large numbers of measurements for performance tuning.

Table of Contents

List of Tables

List of Figures

1 Introduction

Computer designs involving multiple processors — *parallel* computers — have been developed in response to the steadily increasing demand for computational speed. While processor technology continues to advance rapidly, the price to performance ratio of the fastest processors is also increasing. In recent years parallel computers have become more cost-effective for many problems than are uniprocessors of comparable performance [Bailey *et al.*, 1994].

The favorable price-performance value of parallel computers arises from their use of hundreds or even thousands of commodity processors. As a result, the degree to which such *highly* parallel computers achieve their potential for high performance computing at relatively low cost is determined by the efficiency with which their thousands of components are used. The intellectual challenge posed by the attempt to use thousands of processors efficiently is a significant barrier to the adoption of parallel processing as the undisputed paradigm for high performance computing in the future. The difficulty of achieving efficiency is indicated by the large software development costs involved in porting and tuning an application for highly parallel machines (for some typical examples, see [Gustafson *et al.*, 1988; Cheriton *et al.*, 1991; Rothberg and Gupta, 1990]).

This dissertation addresses this barrier to the wider adoption of parallel computing. Our goal is to enable users of parallel computers to more quickly find efficient implementations of their applications. Reaching this goal will allow more efficient implementations to exist (when available development time is limited) and will allow more cost-effective implementations to exist (by achieving a required level of efficiency in less development time).

Although correctness debugging is an essential part of the development process for parallel programs, we do not address debugging in our work. The principles required for efficient debugging of parallel programs are better understood than are those for efficient performance tuning of parallel programs. This seems to be because correct execution is a necessary precondition to efficient execution, and hence initial research efforts naturally focused more heavily on debugging. In addition, research in debugging for parallel programs has developed techniques

such as deterministic replay [LeBlanc and Mellor-Crummey, 1987] that enable programmers to debug parallel programs using methods similar to those used on serial programs. In contrast, techniques for performance tuning of parallel programs have not been able to make significant use of methods developed for uniprocessing, for reasons we discuss in the next section.

1.1 The Serial Performance Tuning Model

The basis for performance tuning on uniprocessors is rooted in the von Neumann machine model. The von Neumann architecture has at least two benefits for performance tuning:

1. The von Neumann machine model suggests how to abstract the execution time of an implementation in simple terms, namely, as the number of instructions executed. The simple relationship between instructions executed and elapsed time encourages algorithms to be analyzed mathematically in advance of implementation. This means that mathematically analyzed algorithms will have some relevance to real implementations, since application performance will differ from the results of analysis by only constant factors.

2. The von Neumann machine model guarantees a simple relationship between the execution time of program components and overall execution time; the execution time of a program is the sum of the execution times of its parts.

These two characteristics of the uniprocessor environment suggest a straightforward approach to performance tuning on serial machines: programmers start from algorithms which can be analyzed for efficiency; proceed to implementations which are known to be big-O optimal (optimal to within constant factors [Knuth, 1976]); and attempt to reduce the values of the constants by measuring programs in execution, identifying which code segments are most cost-effective locations for improvement (*e.g.*, via code profiling [Graham *et al.*, 1982]).

Unfortunately neither of these characteristics holds true in a parallel computing environment. There are two reasons why the serial model of performance-efficient programming breaks down when using a parallel machine:

1. The addition of multiple processors to a computer allows for an immense range of architectural diversity. Parallel computers are distinguished along many dimensions, such as whether processors share memory, how processors communicate, how processors synchronize, and how memory is organized. The wide diversity of machine architectures has made it difficult to determine how to abstract parallel machines for analysis. The main analytic platform, the PRAM [Karp and Ramachandran, 1990], is too far removed

from real machines to provide analyses useful to the implementer. Recent attempts have been made to develop more useful machine abstractions [Snyder, 1986; Valiant, 1990; Culler *et al.*, 1993], but little experience has accumulated as to their relevance. As a result, successful analysis of parallel applications is highly machine specific (*e.g.*, [Kumar and Gupta, 1991]).

2. The simple relationship between the execution time of program components and the program's overall execution time no longer holds. On a parallel machine, the execution time of an application is no longer the sum of its parts, but instead is determined by the application's *critical path* [Lockyer, 1964]. The critical path is the longest sequence of instructions executed by the machine, respecting synchronization constraints across processors and sequential precedence on each processor. As a result there is no guarantee that reducing the execution time of a code fragment will affect the running time of the application.

There are a number of implications arising from the breakdown of the uniprocessor model of application development. First of all, parallel programmers cannot in general start from a theoretical analysis of their application, or from an algorithm known to be optimal. As a result, the most common approach to developing a parallel program is to look for opportunities for parallelism in existing code or algorithms. Unfortunately many algorithms and codes allow for parallelism at multiple levels (such as at the procedure, loop, and instruction levels) [Cybenko *et al.*, 1991; Crowl *et al.*, 1994] and at multiple locations within the program. In addition, there can be a wide range of synchronization methods, scheduling strategies, and task implementations appropriate for each source of parallelism. The result is that when comparing the programming process for a parallel machine to that of a serial machine, the guidance available from theory is diminished, yet the number and range of implementation decisions is increased.

A second result of the lack of theoretical basis for parallel program design is that the most common approach to performance tuning is the *measure-modify* paradigm [Lehr *et al.*, 1989]. In the measure-modify paradigm, programmers first implement a parallel program, aiming for correctness, then measure its execution time to decide whether it performs acceptably. If changes are necessary, the program is modified and the process iterates. There are a number of reasons why measure-modify is an inefficient approach to finding the best implementation of a parallel program. First of all, simply measuring an application's performance gives little insight into the reasons behind any performance problems found. Second, even given an understanding of the reasons behind performance problems, it can be difficult to connect problems with the design decisions leading to them. Finally, and most important, after a new implementation is created, the program may exhibit new performance problems, perhaps even worse than the previous version.

The final implication arising from the breakdown of the uniprocessor model of application development is that the notion of a "best" implementation of any particular program is no longer valid. Variations in machine characteristics, the number of processors used, and even the amount and internal structure of data used by the program can affect an implementation's performance drastically enough that changing any one of these can require modifying the implementation to obtain optimal performance. Thus, unlike the serial case, the process of tuning an application using a fixed number of processors and machine is insufficient for most parallel applications.

1.2 A Parallel Performance Tuning Model

The previous section discussed the gap that exists between analytic and experimental approaches to parallel program development. Analytic approaches are not well enough developed to encompass the wide range of performance effects caused by variation in architecture and application structure. Experimental approaches lack the predictive power to quickly find optimal program designs.

This dissertation describes an attempt to incorporate the strengths of both analysis and experimentation into a set of techniques to speed the development of efficient parallel programs. Our approach is to employ performance measurement to capture all relevant performance effects, and to use performance analysis to generalize those effects into models suitable for performance prediction.

In this dissertation we consider the performance of a parallel application as a function of 1) *external factors,* such as the size of the input data set, the structure of the input data set, the number of processors used, and the type of parallel computer used; and 2) *internal factors,* such as the parallel structure used in the application, the kind of synchronization used, and the compilation techniques and runtime libraries used. The execution time of a parallel application bears a complex and often unexpected relation to all of these factors.

Abstractly, we can consider the programming process as a minimization problem. The objective function is the performance function $T_P(I, E)$ which is a function of the internal factors I and the external factors E. Using the measure-modify paradigm on parallel programs is essentially a way of sampling the value of a program's performance function for particular values of I and E.

Developing efficient programs is currently a much more experimental process on parallel machines than it is on serial machines. The lack of predictive ability available to the programmer in a parallel environment means that a potentially explosive number of combinations of I and E must be examined to tune a parallel application. The search process used by most parallel programmers is expensive and inefficient.

Attempts have been made to solve some of these problems in isolation. For example, work on performance measurement tools for parallel programs (*e.g.*, [Miller *et al.*, 1990; LeBlanc *et al.*, 1990]) has facilitated and improved the process of experimentation used in tuning applications. Likewise, machine-dependent analytic techniques such as scalability analysis [Kumar and Gupta, 1991] have been developed which provide predictive ability for a class of machines and applications. However, a complete solution to these problems requires both the ability to *measure* a parallel application, and the ability to *predict* the performance of the application, as we change external and internal factors. Using a combination of performance measurement and performance prediction, a programmer can both understand the reasons why an application may be performing poorly and understand the potential effects of a change in the program's execution environment or structure.

1.3 Statement of Thesis

The thesis of this dissertation is that a combination of performance measurement and performance prediction can be exploited to converge on an efficient implementation of a parallel program while minimizing the number of alternative implementations considered.

The central feature of our approach is the decomposition of the problem into two problems, which we treat using a unified framework: 1) predicting the effects of external factors and 2) predicting the effects of internal factors.

Our approach is in three parts: 1) we describe how to measure parallel programs in a way that leads to understanding for problem diagnosis and also is useful for performance prediction; 2) we describe an approach to developing performance models of implementations as functions of external factors; and 3) we describe a method for generating performance models of new implementations (that is, new values of internal factors) without requiring that the new applications be executed, or even fully coded. Each of these parts is supported by a tool or tools which automate all or most of each step.

The first part of our work is called *predicate profiling*. Predicate profiling is a method of measuring the parallel overhead present in the execution of a parallel program. Overhead refers to processor time spent performing functions that do not directly contribute to the problem's solution; they are additional operations that are required because the problem is being solved on a parallel machine. Predicate profiling works by assigning the measured overhead to categories (defined by *performance predicates*), which together meet three criteria: completeness, orthogonality, and meaningfulness. Completeness ensures that all the parallel overhead in an execution is captured; orthogonality ensures that no overhead is counted

twice (thus overheads can be summed in a straightforward way); and meaningfulness ensures that overhead categories correspond to states of the execution that have significance in performance tuning. Predicate profiling was implemented initially using a sampling-based mechanism on an SGI Challenge Series multiprocessor, and in its current form is implemented as the tool `pp`, which uses an event-logging mechanism and runs on the KSR1 multiprocessor.

The second part is called *lost cycles analysis*. Lost cycles analysis is the process of assigning a performance model (typically a simple formula) to each of the overhead categories measured by `pp`. The result of lost cycles analysis is an overall performance model for the application as a function of its runtime environment, such as the input data size, the number of processors used, the input data structure, and machine characteristics. Lost cycles analysis is made feasible for programmers by basing the analysis on overhead categories, and by the tool `lca`. The category-based approach means that rather than producing a performance model for the application in total, the programmer only need produce models for the various overhead categories. Category models are typically fairly simple, while the overall application model can be much more complex.

The tool `lca` assists the user in developing category models. The tool: 1) inputs performance data as output by `pp`; 2) selects performance models that are appropriate for the overhead category chosen by the user (along with user-supplied models, if any); 3) fits each performance model to the data in a least-square sense; and 4) outputs the resulting models along with their R^2 values, in text and graphical form. The models selected by `lca` for a particular category are based on the fact that each overhead category has certain frequently-occuring behavior when varying common environmental variables – for example, load imbalance typically follows one of a small set of behaviors as the number of processors is varied. The output of `lca` allows the user to quickly assess and select one from among the likely models for each category. By summing the category models chosen with the help of `lca`, the programmer can quickly construct an overall performance model of the application as a function of the runtime environment.

The third part, *lost cycles modeling*, addresses the final problem of predicting the performance of alternative parallel implementations. We start with the observation that parallel program restructuring usually involves the reorganization of existing code into a new parallel structure – as when serial loops are converted to data parallelism, or multiple subroutines are run in parallel using tasking. We then observe that the performance of code fragments being reorganized can be modeled using lost cycles analysis. Lost cycles analysis is useful because when code is reorganized into a new parallel structure, as a result it is typically run on a different number of processors, or with different input data, etc. That is, different parallel implementations place code fragments in different runtime environments. These observations allow us to decompose the problem of performance prediction of alternative parallel implementations into two subproblems: 1) how

does each code fragment perform? and 2) what is the relationship between the performance of the overall application and the performance of each fragment? Answering these two questions for a (possibly incomplete) parallel implementation we call *lost cycles modeling.*

The first question we can answer directly using lost cycles analysis. To do this we use the tool `pp` to obtain performance measurements of arbitrary code fragments. The output of `pp` is then directly usable by `lca`. The resulting performance model of the code fragment is then stored with that fragment, as a comment in the source.

To answer the second question, we demonstrate a tool-based technique for composing performance models of fragments into a performance model of the entire application. The method is implemented in the tool `lcm`. The tool proceeds as follows: it parses the program source of an application that has been restructured, looking for three things: 1) explicit parallel constructs; 2) explicit data movement; and 3) lost cycles models embedded in comments. `lcm` then composes the lost cycles models for each of the program fragments using an understanding of the parallel constructs and data movement operations. For example, in an application where a tasking directive has been added, it will compose the performance models of each task into an overall model, taking into account the load imbalance induced by unequal task execution time, and properly accounting for the amount of data and number of processors used by each task. In some cases, `lcm` requires additional execution data; for example, to properly predict the effect of task synchronization `lcm` probably needs to trace synchronization operations.

1.4 Outline of Dissertation

The purpose of this dissertation is to show how the three techniques we present can be used to answer a common question asked by a parallel programmer: "What is the best structure for my parallel application?" To do so, we will first review related work, showing how our work builds on and is distinguished from other research (Chapter 2). Chapter 3 will then describe the problems associated with parallel programming and explain why writing efficient parallel programs is difficult. The contributions made by this work are then described, beginning in Chapter 4, which presents predicate profiling. Chapter 5 presents lost cycles analysis, and Chapter 6 describes lost cycles modeling. Finally, we summarize our contributions in Chapter 7.

2 Related Work

Since our work is best described as an attempt to use both performance measurement and analytic modeling to provide performance prediction tools to parallel programmers, we can characterize related work in terms of three areas:

1. **Performance measurement tools and techniques for parallel programs.** Performance tools typically provide a description of an execution in a form likely to explain reasons for poor performance and intended to suggest ways to improve the application. Most such tools don't provide performance prediction features.

2. **Performance analysis techniques for parallel programs.** Parallel program analyses typically study the sensitivity of application performance as a function of the number of processors and problem size. Such analysis is often asymptotic and thus not oriented to predicting the performance of specific applications on specific machines, and usually requires significant programmer effort.

3. **Performance prediction techniques for parallel programs.** Performance prediction attempts to provide predictions of running time, for specific applications on specific machines, usually assisted to some degree by automated tools. Techniques vary depending on the type of input data used. Static techniques use source code (or pseudo-code) as their main input. Dynamic techniques rely mainly on performance measurements, using them to predict other performance measurements.

2.1 Performance Measurement Tools

The majority of the tools and metrics devised for performance evaluation and tuning reflect their orientation towards the measure-modify paradigm. That is, tools rarely provide predictions of performance for executions that have not been

measured. As a result, performance measurement tools typically provide large amounts of information to the user so that the user can understand the application well enough to form informal mental models of the application.

Performance measurement tools, since they are so information-rich, have the difficult task of organizing data collection and presentation in a way that assists, rather than overwhelms, the user. Three basic approaches have been taken.

The *toolkit* approach brings a collection of measurement and presentation methods into a common framework, and allows the user to explore performance data from multiple viewpoints. In the toolkit approach, there is little prior notion of the likely causes of performance degradation; it is typically up to the user to form a conclusion as to which overhead category dominates performance (*e.g.*, data transfer, serial code, synchronization).

The *specialized* approach focuses on one particular overhead category and organizes data collection and presentation in a manner natural for the specific category. In the specialized approach, there is an implicit assumption that the particular overhead category examined by the tool is dominant in explaining performance.

The *characterization* approach focuses on all categories of overhead, and simultaneously evaluates them all for every execution. The characterization approach describes performance problems in terms of categories of overhead, and reports them as such to the user; differences among characterization approaches occur in how categories are defined, and what the category-based information is used for after it is reported to the user. Our work in predicate profiling is an example of the characterization approach.

2.1.1 The Toolkit Approach

Performance analysis toolkits are typically general in scope, and do not present data specifically in terms of pre-defined categories of poor performance.

The PPUTTS toolkit [LeBlanc *et al.*, 1990] is intended to support the entire parallel program development process. As a result, it provides both debugging and performance analysis tools, which are based on instrumented versions of shared-memory synchronization primitives. Data collected is stored in a user-accessible database. Users can use pre-constructed visualization tools on the debugging and performance data, and can write custom queries to examine data in application-specific ways.

The IPS-2 system [Miller *et al.*, 1990] provides additional structure to the performance analysis process by organizing both performance measurement and data presentation in a hierarchical manner. Users can start from statistics gathered at the system level, descend to the application level, and within the application to procedures and individual lines of code. At each level the user can invoke visualizations of performance data. Another information-managing feature of IPS-2

is its ability to draw the user's attention to the critical path (at any level), in schematic form and by profiling the code segments lying along the critical path.

The ChaosMON system [Kilpatrick and Schwan, 1991] is designed to allow the user to specify performance metrics (and derived functions), and presentation of them in graphical form. This is done on an application-specific basis, relying principally on the value of specific global variables in the application program. These variables can be sampled intrusively (synchronously) or non-intrusively (asynchronously). To organize the data collection and presentation, all objects in the system (processes, data sets, graphs) are described used an extended Entity-Relationship (E-R) model. The E-R model applied to an application is sufficiently high-level, yet detailed enough to describe that application for performance measurement. In addition, the E-R description of data sets allows for well-organized data analysis.

Two toolkits that primarily focus on providing multiple presentation methods for performance data are PIE [Lehr *et al.*, 1989] and ParaGraph [Heath and Etheridge, 1991]. In these systems, data collection is handled transparently to the user, and the user's role is to explore various visualizations to discover the most meaningful data presentation method.

2.1.2 The Specialized Approach

In contrast to the general-purpose toolkit approach, the specialized approach looks for performance bottlenecks in a single overhead category. These tools focus the programmer's attention on the code or data that is most to blame for poor performance in a given category. Most of these tools correlate a metric with an application's source lines or procedures, which is called *profiling*. Tools have been built that focus on each of the overhead categories that we consider in our work: memory system effects, insufficient parallelism and load imbalance. synchronization costs, and resource contention.

Memory System Effects

A wide variety of specialized tools have been built that focus on memory system effects, spanning a tradeoff between minimizing tool execution costs and obtaining detailed output. The utility of memory-system tools arises from the increasing sensitivity of modern high-performance processors to the costs of accessing memory. Because it is so important to use the memory hierarchy efficiently in order to achieve good processor utilization, the profiles provide by memory-system tools can frequently help a programmer to increase application performance considerably.

Mtool [Goldberg and Hennessy, 1993] is a profiler for the metric *stall time*. That is, the primary function of Mtool is to measure the time spent by the processor waiting for memory requests to be satisfied. Mtool works by comparing a measured execution of the application with the same execution in a unit-cost-memory simulation. The difference between the two yields the cost of memory access during the execution.

The SHMAP tool [Dongarra *et al.*, 1990] employs a data-centered view by providing a visual representation of memory use. SHMAP presents memory on the screen as a two-dimensional array, and highlights regions of heavily used memory with color coding. The intent is to help the user rapidly find data structures that have high cache miss ratios, by using memory locations as the common frame of reference used by both the application and the measurement tool.

MemSpy [Martonosi *et al.*, 1992] unifies the functions of Mtool and SHMAP, and provides greater detail than either. Since cache misses may occur for a variety of reasons, and because cache misses are often more usefully correlated with data structures than with code sections, MemSpy provides profiles on a code-oriented or a data-structure-oriented basis. It also uses more detailed metrics, such as first reference misses, invalidation misses, and replacement misses. For example, separately reporting on invalidation misses allows the user to determine how much data communication is occurring in a parallel application. The price of this extra detail is much more overhead spent in measurement: while Mtool only requires two executions of the application at full speed, MemSpy simulates the execution in much more detail, and has been reported by the authors to take from 22 to 58 times longer than a full-speed execution.

Finally, PFC-Sim [Callahan *et al.*, 1990] provides detailed information similar to that of MemSpy, but it takes a different approach to obtaining the data. Rather than running the application in detailed simulation, it uses the PFC Fortran source-to-source compiler to instrument reads and writes at the source level. The instrumentation calls execute a memory system simulator. The result is both data- and code-centered output of memory system behavior.

Insufficient Parallelism and Load Imbalance

A number of tools have been developed to profile the parallelism present in an application and focus the user's attention on code segments that contribute to the application's serial fraction [Karp and Flatt, 1990]. Increased serial fraction can occur due to insufficient parallelism, in which the number of parallel tasks created in a code section is smaller than the available number of processors, or due to load imbalance, in which a sufficient number of tasks are created but they execute for unequal lengths of time.

MaxPar [Cybenko *et al.*, 1991] is a tool that computes the dataflow graph for any segment of code and uses its width to determine the average parallelism

available in the code segment, which determines the maximum speedup possible. With each variable, it associates the times it was last written and read. Then, each operation's earliest possible execution time (respecting data dependencies) is computed at run time. The result is a display that shows both execution time and maximum parallelism for code blocks, thereby focusing attention on which code to parallelize. While MaxPar does not directly diagnose parallel code, it assists the user in applying parallelism to the most fruitful portions of the application.

SPAN [So *et al.*, 1987] is a tool that measures effective parallelism for an entire application. The program is instrumented to record the points where there is synchronization between threads. Then the program is run on a uniprocessor, and a task trace is produced. The task trace saves data including the execution time of each task; the tasks in the trace are then scheduled based on the real scheduling policy of the multiprocessor of interest, producing a measure of the average parallelism in the execution of the application.

In contrast to MaxPar and SPAN, Quartz [Anderson and Lazowska, 1990] operates on parallel applications and provides detailed measurements of actual parallelism. Instead of processing time per procedure, Quartz measures *normalized processor time* (NPT), which is defined by:

$$NPT = \sum_{i=1}^{P} \frac{\text{processing time with } i \text{ processors concurrently busy}}{i}$$

where P is the number of processors. The NPT metric shows the concurrent effective parallelism while a routine executes. This metric is based on the observation that serial code that always executes concurrently with other computation will have little effect on performance compared to a serial routine that always executes by itself. The benefit that Quartz provides as a profiling tool is to provide a measure of the performance gain to be expected from optimizing a particular section of serial code – similar to the benefit provided in a serial environment by the execution-time profiler Gprof [Graham *et al.*, 1982].

Synchronization Costs

A tool for profiling synchronization costs is described in [Davis and Hennessy, 1988]. By instrumenting the primitives used to incorporate parallelism into the application, synchronization events are traced at runtime with low overhead. The profiles are presented based on synchronization objects such as locks and mutexes. The metrics profiled include synchronization operation frequency, execution delay, number of processes concurrently attempting access to a lock, amount of time between synchronization operations, and time spent holding a lock. Runtime measurement increases application runtime by only a few percent, although lock acquire time can increase by a factor of ten. The result is an assessment of the

performance impact of synchronization operations on the application, and data indicating which synchronization objects are most responsible for the performance impact.

Resource Contention

Resource contention occurs when high demand is placed on a system component with limited service capacity, such as a memory module, an interconnection network, or a communication channel. Although the pp tool we present in this dissertation is the only one we know that directly measures resource contention, previous work has shown how contention effects can be measured. In [Tsuei and Vernon, 1990], the authors describe measurement methods for resource contention, which are based on measuring mean inter-reference time and mean memory cycle time, and comparing to known ideal times for both. The method used can provide a summary of the total time spent waiting for resources (bus and memory) by the application. Unfortunately, the approach described requires the use of additional hardware to monitor the system's bus. The approach we use does not rely on additional hardware, but makes use of the dedicated performance monitoring hardware on the KSR series multiprocessors.

2.1.3 The Characterization Approach

Specialized tools are very useful in application fine-tuning, but usually do not provide completeness (*i.e.*, they don't measure *all* sources of overhead in the execution). As a result, they are limited to cases in which the principal overheads are known in advance.

A number of researchers in the parallel performance evaluation and tuning community have focused on measurement of multiple parallel overheads. In particular the PEM system has developed a taxonomy of parallel overheads similar to ours [Burkhart and Millen, 1989]. Their measurements, while providing similar overall data to the user, are not oriented toward performance prediction of alternative implementations, so the specific overhead categories they use are not always amenable to easy analysis. In addition, they have not emphasized the completeness criterion, which is necessary to use measured data as a basis for performance prediction.

In [Møller-Nielsen and Staunstrup, 1987], the authors define the problem-heap model, in which a number of tasks ("problems") are created and placed in a heap. Tasks are then individually removed from the heap and executed by processors as they become available. The authors characterize sources of performance loss due to this particular software organization:

1. Starvation loss, which occurs when there are not enough problems in the heap. This can happen especially at startup time, and corresponds to our "insufficient parallelism" category.

2. Braking loss, which occurs when tasks are still running after an answer is found. This is a category we used in our first implementation of pp, and measured for the subgraph isomorphism problem described in Section 3.1.

3. Separation Loss, which occurs when then splitting of a task into two may make more work. For example, in parallel string search, the beginning of the string being searched may be searched twice, once before and once after the split. This category is represented in our work as wasted computation, another overhead used in our study of subgraph isomorphism.

4. Saturation loss, which occurs due to the mutual exclusion necessary on the heap. This category is included in our synchronization loss category.

The overhead categories presented here are restricted to software effects only, and hence do not provide completeness.

Two tools that define sources of poor performance in advance of execution are the W3 system and ATExpert. The W3 system [Hollingsworth and Miller, 1993] defines sources of poor performance in advance of execution, but the categories they use are focused on automated diagnosis and do not have a role in performance prediction. The tool ATExpert [Kohn and Williams, 1993] provides both multi-category performance measurement, and some simple performance prediction in the form of expected speedups for parallel loops. However it does not address performance prediction in a general way. It cannot handle arbitrary program structures, and relies heavily on specific knowledge of the Cray performance measurement hardware and automatic loop parallelization software; hence it is difficult to generalize from the ATExpert tool to other platforms.

2.2 Parallel Performance Analysis

As discussed in Section 1.1 efficient development of parallel programs must be based on some predictive component, since the measure-modify approach is efficient only for a small number of implementations, inputs, or execution environments.

Any ability to predict the performance of an application in advance of execution requires a model of the application's performance. Models can be developed in two ways: they can result from an analysis of the algorithm or program — *static* analysis; or they can result from inference based on known values of the application's performance function — *dynamic* analysis. Static analysis is a systematic

approach that relies on complete understanding of the machine, application, and input; dynamic analysis is an empirical approach, using the machine, application, and input to demonstrate directly the performance effects that are important.

Although analysis provides the conceptual tools to predict parallel program performance, most previous work in analysis has not been directly used by programmers to predict performance of real applications for two reasons:

1. Analysis techniques have primarily focused on the study of *asymptotic* application performance. The results typically describe upper bounds on the application's efficiency to within a constant factor. However, alternative implementations of a program may often have the same asymptotic performance function, yet differ in important ways in the values of the associated constants.

2. Analysis techniques are typically the products of careful study and significant effort. The work required in developing an analytic model can greatly outweigh the effort in simply implementing and measuring a proposed alternative program structure. In addition, the difficulty involved in developing analytic models of overheads means that often, some overhead categories are not considered. The overlooked categories may be insignificant for the applications studied, but it can be difficult to generalize the results to other applications for which the overlooked categories may be significant.

Overcoming these two problems is the difference between parallel performance *analysis* and parallel performance *prediction*. We refer to work that does not attempt to model actual values of running time (instead focusing on "operation counts" such as data items transferred through a network, comparison operations, or floating point operations), and is performed primarily by the user, as parallel performance analysis. Work that models actual running time of applications, and attempts to automate the process of developing such models, we will refer to as parallel performance prediction. The remainder of this section reviews the work in parallel performance analysis; the next section reviews prior work in parallel performance prediction and contrasts it with our approach.

While serial analysis concentrates on how performance changes as a function of input data size n, parallel performance analysis generally develops functions of both n and of the number of processors used, p. In addition, parallel performance prediction must take into account many more independent variables, since they can affect performance. We show in Chapter 3 that an application run for fixed values of p and n can still have significant performance variations as one varies the structure of the input, the problem definition, or the machine type.

A very large body of work has analyzed algorithms in idealized machine frameworks, such as the PRAM [Karp and Ramachandran, 1990], that are essentially

unrealizable because they ignore important sources of overhead in real machines. In order to model applications running on real machines, parallel performance analysis techniques must do more than model the productive operations in a computation; they must also model the non-productive, or overhead operations. As mentioned in Section 1.1, attempts have been made to develop more useful abstract machine models, in which realistic overheads are incorporated [Snyder, 1986; Valiant, 1990; Culler *et al.*, 1993]. However, little experience has accumulated to date as to the ability of abstract machine models to accurately model real machines; hence the parallel performance analysis techniques we review here are all performed, in practice, in the context of specific overheads and specific machine architectures.

Within the wide spectrum of parallel performance analysis, relevant work includes the body of literature that has studied how parallel overheads behave — that is, what typical functions can be used to describe parallel overheads. The large body of performance analysis that applies queueing networks and timed Petri nets to the modeling of parallel systems has not generally focused on understanding overhead categories[1] and so falls outside the scope of this dissertation. However, a number of papers have studied parallel overheads in general, while many papers have studied specific parallel overheads.

2.2.1 General Studies of Parallel Overhead

Papers that have studied parallel overhead in the abstract have tried to identify techniques that can assist programmers or system designers in predicting optimum values for n or p. The total overhead in an execution is commonly denoted $T_o(n,p)$, while the non-overhead or "pure" computation is denoted $T_c(n)$. We denote the execution time of the parallel application as T_P. Using these terms, the observed execution time of the parallel application is:

$$T_P(n,p) = \frac{T_o(n,p) + T_c(n)}{p}.$$

Two metrics commonly studied in these terms are *speedup*:

$$S(n,p) = \frac{T_c(n)}{T_P(n,p)}$$

(in which we assume that we are parallelizing the best known serial algorithm) and *efficiency*:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_c(n)}{pT_P(n,p)} = \frac{T_c(n)}{T_o(n,p) + T_c(n)}.$$

[1] With some exceptions — *e.g.* work that attempts to extend queueing analysis to include software structure along with contention effects is described in [van Gemund, 1993].

In [Eager *et al.*, 1989], the authors are concerned with the relationship between speedup and efficiency, and in particular how to predict the point at which the greatest ratio is obtained of efficiency over execution time. The point at which this ratio is obtained represents an optimal operating point for the application, since the application is utilizing processors efficiently (using system resources well) while achieving low execution time (using the user's time well). They show that, considering parallel overhead to consist only of load imbalance, the *average parallelism* of the application — the number of processors usable by the application at each instant, averaged over its execution time — is the point at which the efficiency / execution time ratio is maximized.

A similar goal is pursued in [Flatt and Kennedy, 1989], but additionally the analysis includes communication and synchronization components of $T_o(p)$. The authors present a number of general assumptions about parallel overhead, which 1) ensure that the derivative of running time has at most one zero – thus, if running time hits a minimum, it never subsequently decreases; and 2) ensure that eventually the running time will begin to increase (hit a minimum). They show that overhead functions that are logarithmic (the best possible) or linear in p, among others, fit these conditions. They also develop, in their framework, expressions for the same operating point used by [Eager *et al.*, 1989]. They develop T_o functions for some common software structures (schedulers that incorporate critical regions; barrier synchronization time in hypercubes and MINs) and then develop optimal operating points for these structures. Finally, they consider scaled speedup [Gustafson, 1988] and show (assuming T_o is independent of n) that "Kuck's conjecture" holds, namely that if one multiplies both n and p by k, scaled speedup can increase at most by a factor proportional to $k/\log(k)$.

Even more overhead types are considered in [Carmona and Rice, 1991], which refers to parallel overhead as *wasted work*. Included in wasted work is load imbalance, communication, contention, task activation/termination, and synchronization. The authors generalize the notion of serial fraction [Karp and Flatt, 1990] by making it dependent on both n and p, and show that besides just serial code, an application's serial fraction could also be wasted work that increases linearly with p.

The limitation of studies that do not consider specific applications and machines is that they can only provide general techniques for determining optimal operating points of an application; there is still significant effort required for any particular application in applying those techniques. A number of studies therefore have analyzed particular applications as an aid to programmers who may be interested in creating similar implementations. For example, Nicol and Willard [1988] study the solution of elliptic PDE's using iterative methods on hypercubes, grids, and bus architectures. They derive expressions for running time and explicitly model communication costs. The resulting expressions are used to determine optimal problem size or machine size, and to show how optimal speedup increases

as a function of problem size on different machine types.

Scalability analysis [Kumar and Gupta, 1991] develops analytic, asymptotic models of computation and selected overhead categories as a function of the size of the problem n and the number of processors p. Many studies of the overheads in specific applications have been performed within the framework of scalability analysis; examples include [Grama and Kumar, 1992; Singh *et al.*, 1991; Kumar *et al.*, 1994]. These studies provide insight into the inherent scalability of a particular application and machine combination. The distinguishing feature of scalability analysis is in its comparison of an application's efficiency as a function of p with its efficiency as a function of n. The ratio of $E(n)$ to $E(p)$ provides an understanding of how an application's problem size must be scaled up as the number of processors used is increased, and is called the application's *isoefficiency* function [Grama *et al.*, 1993].

Scalability analysis and related techniques provide useful insight into the strengths and weaknesses of applications on specific parallel machines. However, when using scalability analysis it is necessary to anticipate the overhead category or categories that will dominate in actual practice. The notion of completeness in our work attempts to address this weakness of scalability analysis by consistently measuring and modeling all categories of parallel overhead.

Finally, an approach that combines measurement of overheads broken down into categories with modeling is presented in [Sivasubramaniam *et al.*, 1994]. The authors use simulation to study three kinds of parallel overhead: application-level. communication, and contention, and generalize the observed data using mathematical models. The use of simulation allows the authors to study alternative architectures, and to use real aplications, but limits their conclusions to scalability observations rather than performance predictions.

2.2.2 Specific Studies of Parallel Overheads

Scalability analysis can provide understanding of the general kinds of functions that parallel overheads might follow, which provides boundaries for the model-selection process used in the tool `lca`. However, the specific functional forms that each category of parallel overhead is likely to follow form the basis for the default functions used by `lca`; work that has studied those specific forms is reviewed in this section.

Many studies have developed analytic models of the cost of communication and resource contention for particular application / machine combinations. Typical examples are: [Brochard and Freau, 1990], which models communication costs in the IBM RP3; [LeBlanc, 1988], which models data movement costs both due to locality and due to algorithmic communication requirements, for Gaussian elimination on the BBN Butterfly; and [Crovella *et al.*, 1992] which models

communication costs and resource contention for Gaussian elimination and parallel Quicksort on two models of the BBN Butterfly and the Silicon Graphics 4D/480 multiprocessor. Each of these papers starts from a careful study of the application's communication pattern, and identifies the types of communication that occurs. Each also measures directly the costs of each type of communication present.

A somewhat more automated approach is the use of *load/store kernels.* Load/store kernels are small code segments that repeatedly perform common memory operations, allowing accurate measurements to be made of the costs of communication, local memory operations, and interconnect performance. Load / store kernels are particularly useful on shared memory machines used for data-parallel applications. In this environment, load imbalance, insufficient parallelism, and synchronization loss often are all minimal, so that memory and interconnect characteristics dominate performance. Unfortunately, as with many performance analysis techniques, the load/store kernel approach seems to require significant user effort in practice.

For example, in [Gallivan *et al.*, 1991] the authors describe a method for predicting the effects of memory access on the performance of data parallel Fortran codes on the Alliant FX/8. Load/store kernels are used to characterize the machine in detail, including the length of vectors processed, the load/store pattern, and memory request density. Performance of the machine over the parameter space, using an adaptive sampling resolution, is stored in a database. Code samples are matched to the closest (interpolated) measurement in the database as follows: First, code is converted to a pattern of NOPs, LOADs, and STOREs. Next, the Load/Store pattern is matched to one in the database. Finally, the memory reference density and miss ratio are estimated. Much of the process seems to be performed by the user, but it leads to performance predictions of loop timings which are quite accurate — agreeing to within about 10 percent.

A related approach was taken in [Bodin *et al.*, 1990], which models communication costs on the BBN GP1000 using load/store kernels [Gallivan *et al.*, 1991]. Instead of storing the results of the load/store measurements in a database, the measurements were used to parameterize an analytical model of network-connected memory system performance which includes memory contention. The model predicts the performance of a simple application with reasonable accuracy (within 25%); the remaining differences are assumed to be due to network contention.

Work that includes models of synchronization costs includes [Nanda *et al.*, 1991] and [Tsuei and Vernon, 1990]. These studies include models of contention for software structures, which appears as waiting time to acquire locks. In [Nanda *et al.*, 1991], the authors parameterize the application space into six dimensions: 1) the number of concurrent processes, 2) the number of shared-data objects, 3) the number of local computational operations between critical sections, 4) the

number of shared data references between critical sections, 5) the amount of time spent inside each critical section, and 6) whether the workload generated from these parameters is fixed or probabilistically generated. Using workloads spanning this space, the paper takes measurements of overhead due to: 1) network latency, 2) hardware contention, 3) software (lock) contention, and 4) time spent in lock/unlock routines on on a Butterfly Plus (GP1000), a Bfly-II (TC2000), and a Sequent Balance 21000. Analytic forms of these overheads are then developed. The result is a prediction of the communication, resource contention, and synchronization overhead to be expected for an application whose characteristics along the 6 dimensions are known, running on one of the three machines used. Thus, this approach is based on a parameterization of applications along a number of dimensions sufficient to characterize the application's overheads. Open questions remain as to whether real applications are sufficiently uniform in general to be characterized by a small set of measurements, and how to extend the measurement dimensions to include remaining overhead categories, such as load imbalance and insufficient parallelism.

Another approach at modeling synchronization, communication, and resource contention is taken in [Tsuei and Vernon, 1990]. The authors break speedup limitations into 3 kinds: software structure, lock contention, and hardware contention. They argue that for common kinds of programs (SPMD) that these 3 limitations are not circularly dependent; thus efficiency ratios due to each can be simply calculated and are multiplicative. The software structure factor is obtained by measuring the application's execution profile (efficiency for various values of input size and number of processors). Models of lock contention are based on queueing network mean value analysis [Lazowska *et al.*, 1984] using the mean lock inter-request time, the number of locks, and the time spent holding each lock. These inputs depend on the execution profile. Hardware contention is modeled by actually measuring mean inter-reference time and mean memory cycle time, and comparing it to ideal values for memory cycle time; this step requires a bus monitor. The resulting models provide accurate performance estimates for various versions of Gaussian elimination. While the models for synchronization costs used in this work are not applicable to arbitrary parallel programs, the use of simple cyclic queueing models to approximate contention effects is widely applicable.

Load imbalance and insufficient parallelism have also been modeled for specific applications. In [Zhang and Srinivasan, 1990], the variation among task running time is measured, and used to select a scheduling strategy. The paper points out that when variation in task running time is identically distributed, then load imbalance should increase with the square root of the number of processors (a result from order statistics).[2] When the standard deviation of task running times is large, then tasks should be self-scheduled — otherwise, tasks should be pre-

[2] A Pth order statistic is the maximum of P identically-distributed random variables. A distribution-free bound on the expected value of a Pth order statistic (the Pth-smallest value)

scheduled. Two applications are shown: the first has high variation in task running time, and the load imbalance predicted indicates that the application favors self-scheduling. The second has nearly constant task running time, and hence favors pre-scheduling due to its smaller synchronization costs. Another study that uses similar order statistics to model load imbalance is [Hummel *et al.*, 1992].

2.3 Parallel Performance Prediction

While the performance analysis techniques reviewed in the previous section are useful for specific applications on specific machines, they do not represent a general approach to performance prediction. To be useful to parallel programmers, performance prediction must result in predictions of actual running time, must be as automatic as possible, and must be applicable to most or all parallel programs and machines (and hence cannot overlook any categories of parallel overhead).

Any performance prediction technique must include a static analysis component and a dynamic analysis component. Static analysis is the foundation of prediction: the programmer considers a program structure and desires to know what its performance might be. Dynamic analysis captures the costs specific to the particular runtime environment and machine used (since source code is ideally useful on multiple machines).

Static analysis techniques operate on a wide range of source code formats. Some techniques assume a particular parallel programming language or programming model. Other static analysis techniques require that the programmer specify a *template* for the application. In a template approach, the programmer identifies a commonly-used parallel structure, such as producer/consumer or central task queue, and associates program components with the logical components of the template.

Dynamic analysis techniques rely on performance measurements to characterize the runtime behavior of applications, but must attempt to minimize the number of measurements taken. To avoid completely measuring the application's performance function (and thus defeating the purpose of performance prediction) dynamic techniques must infer the application's performance function from a restricted set of measurements. Usually static analysis provides some basis for the inferences made. Dynamic techniques vary widely in the amount of data they collect; from measuring multiple executions per application, to measuring a single execution per application. to taking a set of measurements only once per machine.

Performance prediction techniques all incorporate static and dynamic components, and form a spectrum from highly static with little dynamic measurement, to

---

of identically-distributed random variables with mean μ and variance σ is $\mu + \sigma\sqrt{P/2}$ [Gumbel, 1954; Hartley and David, 1954].

highly dynamic with a small static component. The central issue for a performance prediction technique anywhere on this spectrum is the method of *automating* the static and dynamic components. The main difficulty experienced by highly static approaches is the inability to predict the behavior of the application at runtime, based on its source code. These problems are similar to those that are presented to compiler writers seeking opportunities to apply optimizations or data dependency analysis. The main difficulty experienced by highly dynamic approaches is efficiently and reliably forming inferences from performance measurements. This problem is similar to the "function discovery" problem: data points may suggest a certain function, but incorrectly so due to aliasing arising from subsampling. Gathering enough data to conclusively verify a performance function in the presence of noise and measurement error can be too expensive to provide a viable basis for performance prediction.

2.3.1 Static Approaches

Research that relies almost exclusively on static analysis is presented in [Clement and Quinn, 1993]. Using the Dataparallel C source-to-source compiler, estimates are made directly from the source of the number of serial instructions, parallel instructions, uncached memory accesses, runtime-parallelization overhead, and number (both not size) of messages sent. Under the assumptions that programs are being run on multicomputers with wormhole routing, high message startup costs, and similar instruction times for floating-point and integer instructions, only a few machine-specific measurements are required to parameterize the resulting models. While good predictions can be made for some codes, other codes (with more dynamic runtime behavior) are harder to predict accurately.

Other approaches that emphasize static analysis simplify the problem by not providing a complete performance prediction capability. For example, in [Hickey *et al.*, 1992] the authors use dynamically-captured synchronization information along with statically-analyzed program structure to predict the performance of Ada applications on varying numbers of processors. However, this work does not assess the effects of varying problem size, which are more difficult to predict statically than are the effects of varying the number of processors used. Another technique is described in [Fahringer and Zima, 1993], which uses the source analysis capabilities of the Vienna Fortran Compilation system along with a single instrumented run to predict statistics such as amount of data transferred and network contention time. However these statistics are not directly translatable into predictions of execution time, so they are used to infer the relative performance of alternate versions of an application.

2.3.2 Training Set Approaches

A larger collection of machine-specific information is used by techniques employing *training sets*. Training sets are collections of operation sequences that capture all the relevant performance characteristics of a parallel machine; these sequences are measured and values are stored in a database for use by the prediction tool. For example, a training set for a multicomputer may include gather and scatter operations, global synchronization operations, and point-to-point communication. Training sets represent the extension of the load/store kernel concept to include *all* machine operation sequences that can incur parallel overhead. The benefit of the training set approach is that complex machine characteristics resulting from parallel execution can be captured by the dynamic component of the technique (training set measurement) allowing the remaining static analysis to be much simpler, and based more fully on principles of serial code analysis.

The training set approach is taken in [Balasundaram *et al.*, 1991], which combines the cost of typical message-passing patterns (such as broadcast and message-shift-left) with the known costs of serial code. This approach is restricted to SPMD programs and requires the user to supply values that are difficult to statically infer, such as the values of loop bounds. The system achieves good accuracy for programs run on an NCUBE.

A tool that also employs extensive machine-specific data is the PAWS system [Pease *et al.*, 1991]. PAWS statically analyzes Ada code, converting it to an intermediate, dataflow representation. It then maps the dataflow graph onto a specific machine and uses detailed characterization of the hardware to calculate speedup. A large number of architectural parameters are needed to support this method, and the method's accuracy isn't presented.

Extensive machine data is collected in [Fahringer, 1994], and many difficulties of the training-set approach are discussed there. The paper points out that training sets are not portable in general, and not only need to be machine specific, but also compiler specific. The performance of kernels when occurring in actual applications was often different from their performance measured in isolation, due to register allocation, cache, and CPU pipeline status. The need for a large number of fairly large kernels was identified, which makes the pattern matching problem difficult. Finally, the paper concludes that a large effort is required in collecting, verifying, and maintaining a training set.

2.3.3 Template Based Approaches

The template approach is popular because it simplifies both the static analysis and the dynamic analysis components of performance prediction. Templates subsume most of the static analysis task in the selection of a template — complex

performance models are constructed in advance and implicitly selected when a template is chosen. Likewise, a reduced number of dynamic measurements need be taken to parameterize the template's model. Unfortunately, the template approach restricts the range of implementation options available to the programmer, and requires the programmer to express the application in terms of a template. In addition, significant effort is required in developing the models for a template on a particular machine, but minor variations in the structure of a template can invalidate the models derived from them.

The work that first applied the template approach is the PERMOD system [Zimran *et al.*, 1990; Vrsalovic *et al.*, 1984; Vrsalovic *et al.*, 1988]. In the PERMOD approach, templates are referred to as Implementation Machines (IMs). The goal of the PERMOD approach is to select the proper IM based on comparing analytic expressions of the application's performance function for each IM. The methods are applied to a PDE solver using Jacobi's method on a Sequent and on simulated distributed systems. On the Sequent, the results differ somewhat from prediction because some the IMs are fairly sensitive to scheduling irregularities and load imbalance.

In [Sreekantaswamy *et al.*, 1991], the template approach is applied to applications running on T800 transputer-based multicomputers. The templates used are a "processor farm" (a central work queue with no hierarchy of tasks) and the "divide and conquer" style (in which tasks can be created hierarchically). The machine parameters used are the communication cost of creating a task, and the communication cost of starting a task, which are measured in advance of model construction (task scheduling is supported in hardware on the T800). The model outputs throughput measured in tasks executed per second. Using this output metric, actual task execution time need not be measured (all tasks are assumed to require equal execution time, and have equal communication requirements). Models are built up from the leaves of the computation as recurrence relations. Under these assumptions, the error in prediction is on the order of 5 percent. Uses of the models are: 1) predicting maximum speedup and 2) predicting change in speedup with change in granularity (by grouping tasks).

The template approach is used in [Nudd *et al.*, 1993] to overcome difficulties in applying Software Performance Engineeing (SPE) [Smith, 1990] to parallel program development. SPE is a method of specifying applications in advance of implementation that allows the derivation of performance estimates. The authors point out that applying SPE to parallel systems has significant difficulties that can be addressed by using templates to capture application-dependent portions of the SPE specification. They show how an image processing program can be specified using templates as input to SPE, and demonstrate good agreement between prediction and actual performance for their example application.

2.3.4 Dynamic Approaches

As mentioned above, dynamic approaches have the advantage that the application's runtime behavior (loop bounds, communication patterns, etc.) are directly observable, but they have the difficult task of generalizing performance functions from the observed data. Thus, dynamic approaches are distinguished by the methods they use to construct performance functions.

A number of approaches use data *clustering* to reduce performance data to a compact representation. In [Abrams *et al.*, 1992], the authors describe a system that helps the user develop a semi-Markov model of program state evolution. The tool starts with a raw state trace, and allows the user to aggregate and eliminate events until a small set of events remains, from which an empirical semi-Markov model is formed. The tool provides visualizations of state traces, in both time and frequency domains, to aid in trace reduction. For example, the frequency domain is useful because patterns exposed there are subject to aggregation transforms.

Another clustering approach is described in [Dimpsey and Iyer, 1991]. The state of the hardware and operating system is characterized by a number of parameters (*e.g.*, run queue length). The application is run repeatedly over a long time frame, and system states are statistically clustered to become states in a Markov model. The state transition probabilities are established empirically. The resulting Markov model can be solved via Monte Carlo simulation to obtain a predicted probability distribution of the running time of the application. The results agree well with the measured distribution for the application presented.

Highly empirical clustering techniques such as those described in [Abrams *et al.*, 1992] and [Dimpsey and Iyer, 1991] are characterized by many design decisions (initial selection of system state parameters, the appropriate number of clusters) whose proper solution is not generally obvious. These methods require substantial effort for a single instance of a program on a specific input set. In addition, it's not clear how to use the resulting model in performance prediction since it may not be understood what aspects of the model remain valid when any part of the system is changed.

A performance prediction approach applying moderate static analysis along with moderate dynamic analysis to SPMD programs is presented in [Mehra *et al.*, 1994]. The tools described parse the node program's source to determine syntactic structure. The structure is then matched using regular-expression pattern-matching to values obtained from trace files to get constants and formulas associated with each parse node. A model of the node program is then built which can be replicated and used in simulation, to predict the communication costs and synchronization effects of message passing.

One difficulty in applying static analysis is that large, complex expressions result from the analysis of real applications. Work described in [Clement and

Quinn, 1994] outputs symbolic expressions for operation counts (such as communication operations and loop iterations) in a form suitable for use by the Maple mathematical manipulation system [Char *et al.*, 1991]. A single instrumentation run of the application captures much of the needed dynamic information (*e.g.*, hard-to-predict loop iteration counts) and remaining dynamic information is estimated. By using the Maple system, performance predictions can be easily simplified, and then cast into a wide variety of forms, such as predictions of execution time, speedup, sensitivity to system parameters (such as communication cost), and optimal operating points for the application.

2.4 Relationship to Other Work

The work described in this dissertation builds on and can be contrasted with prior work in performance measurement tools, performance analysis, and performance prediction.

The predicate profiling technique we describe in Chapter 4 stands out from most prior work in performance measurement in attempting a high-level, but complete characterization of application performance. In addition, the notion of specifying the causes of poor performance in advance of execution is present in few other tools — notably, the W3 system [Hollingsworth and Miller, 1993]. We differ from that approach in focusing on a dual role for our measurements — performance evaluation and performance prediction.

The performance prediction work described in Chapter 5 (lost cycles analysis) draws heavily on work done in performance analysis. Lost cycles analysis is almost entirely dynamic; we show in that chapter that prediction of the performance function for a single application can be based on very simple programmer analysis. The leverage we use to achieve simplicity for the programmer is the knowledge of typical behaviors of parallel overheads, as reported in the literature. Lost cycles analysis starts from the observations of Flatt and Kennedy [1989] that overhead has certain typical characteristics. We additionally use the even more specific existing knowledge about typical overhead behaviors described by the papers reviewed in Section 2.2.2. Breaking overhead into categories, and using specific knowledge of how different categories of overhead behave allows us to avoid statically developing performance models for parallel overhead from scratch.

The work on performance prediction described in Chapter 6 is a static approach that is similar in some ways to [Clement and Quinn, 1994] in its symbolic manipulation of performance models. However the basis of our approach is the observation that models can be associated with code fragments, and manipulated simultaneously with their associated source code, a feature which is absent from prior symbolic manipulation work. In addition, the simplification of the static analysis task resulting from the addition of embedded performance models in the

application is a significant departure from prior work in static performance analysis.

In summary, our performance prediction approach starts from the use of mainly dynamic data (lost cycles analysis) and incorporates simple static analysis (lost cycles modeling). Our approach differs from other approaches in separating the performance problem into two parts: prediction of external effects, and prediction of internal effects. We gain useful leverage on the problem in this way, since the prediction of internal effects can make use of external effect prediction in its solution.

After separating the performance prediction problem into parts, we show that each part is most appropriately solved by a different approach. Predicting the performance of an application as a function of varying external factors is solved by a mainly-dynamic approach, since it is particularly difficult to apply static analysis to this part of the problem. Predicting the performance of an application as it is restructured is then solved by a fully static approach, which allows us to develop predictions without the need for new dynamic data measurement.

3 Difficulties in Performance Prediction

Writing efficient parallel programs is difficult because the running time of a parallel program is dependent on many factors, in complex ways. In order to enable parallel programmers to quickly find the best structure for a parallel program, it is important to understand the range of factors that can affect performance. This chapter enumerates and classifies those factors. The analysis presented in this chapter serves to clarify the problems associated with parallel programming for efficiency, and lays the groundwork for the solution we present in later chapters.

We divide the factors affecting performance into two groups: *external* factors, and *internal* factors. External factors are aspects of the program's run-time environment not specified in the program's source. In our work we consider 5 external factors:

1. the number of processors used to run the program;
2. the size of the input data set;
3. the structure of the input data set;
4. the problem definition; and
5. the machine used to run the program.

The structure of the input data set refers to the contents of the input data, *e.g.*, the sparseness of an input matrix. The problem definition refers to programs that can solve multiple variations of a problem, *e.g.*, a program that runs until convergence to a user-specified value.

Internal factors are the methods used to parallelize the application. These correspond to changes in the program's structure. Typical examples of internal factors are:

1. the type of parallelism used: *e.g.*, task parallelism, data parallelism, or vector parallelism;

2. the choice of which code to parallelize: *e.g.*, how many of the program's loops to parallelize; and

3. the choice of synchronization methods: spinning vs. blocking, or locks vs. barriers.

Many more internal factors could be listed here.

The remainder of this chapter will show that each of the external factors listed here can affect the choice of how best to parallelize an application. That is, the specific values of external factors determine the best choice of internal factors. This means that the programmer cannot expect in general to find the one "best" structure for a parallel program; the proper structure must be chosen as a function of the run time environment. Our work is motivated by the observation that the measure-modify approach to program tuning cannot cope reasonably with programs whose performance is a complex function of its run time environment. A goal of our work is to show that performance prediction can be used effectively to tune parallel programs for which measure-modify is impractical.

To illustrate the dependence of program performance on the combination of internal and external factors, we will present two cases studies. The first, parallel subgraph isomorphism, is a symbolic application that shows the wide range of external factors that can affect the choice of best program structure. The second, parallel 2D FFT, is a more regular, numerical application that shows the need for performance prediction as a function for machine, number of processors, and input data size.

3.1 Subgraph Isomorphism

An example that demonstrates the difficulties posed by multiple implementations is parallelizing an algorithm for the *subgraph isomorphism* problem. Given two graphs, one small and one large, the subgraph isomorphism problem is to find one or more isomorphisms from the small graph to arbitrary subgraphs of the large graph. An isomorphism is a mapping from each vertex in the small graph to a unique vertex in the large graph, such that if two vertices are connected by an edge in the small graph, then their corresponding vertices in the large graph are also connected. The basic algorithm we use organizes possible solutions into a tree, and searches the tree for actual solutions. Subgraph isomorphism is NP-complete, but by applying *filters* at each node of the search tree, large portions of the search space can often be pruned, allowing solutions to be found in a reasonable amount of time.

This algorithm has a number of potential parallelizations[1] including:

Tree parallelism searches subtrees of the root node in parallel;

Loop parallelism parallelizes the loops within each filter (since all the filters work by iterating over the nodes in each graph);

Instruction parallelism packs graph connectivity data as bitmaps into words, allowing set intersection operations to be implemented as Boolean operations within the filter loops.

We created a program to solve subgraph isomorphism that can implement tree, loop, and instruction parallelism, in any combination. Thus our program incorporates 8 different implementations.

Our program runs on 7 shared-memory multiprocessors: the Sequent Balance, the Sequent Symmetry, the Silicon Graphics Challenge Series, the BBN Butterfly, the BBN TC2000, the IBM 8CE, and the KSR1. All of these machines have at least 8 processors; on some machines we used as many as 32 processors.

Input to the program consists of two graphs, generated randomly. Based on the random process used to construct the two graphs, we can estimate the probability that any given leaf node in the search tree represents a valid isomorphism, which we call the *density* of the solution space. When the small graph has few edges and the large graph has many edges, the solution space is dense; when the small graph has many edges and the large graph has few edges, the solution space is sparse.

The program can search for any number of isomorphisms; in our experiments we vary the number of solutions requested from 1 to 256. We refer to this as varying the *problem*, since the implementation and input are fixed.

Different parallelizations have widely differing performance as a function of machine, number of processors, input, and problem. The performance of each parallelization is a function whose domain is this 4-dimensional space. Problems requiring selecting among the various parallelizations can come in many forms:

- for a fixed machine, number of processors, and problem, we may need the best parallelization as we vary the input density;
- for a fixed machine, number of processors, and input density, we may need the best parallelization as we vary the problem;

[1] Other parallelizations are also possible, including vector parallelism (vectorizing the iterations within filter loops) and filter parallelism (applying filters at each node in parallel); however our experiments do not include them.

Table 3.1: Comparison of Loop and Tree Parallelism in Varying Environment

Density		8CE	Butterfly	Balance	SGI	Symmetry	TC2000	KSR1
10^{5}	Loop	24.7	75.7	91.7	2.06	29.7	11.0	10.7
	Tree	36.1	12.6	86.7	2.59	15.9	3.79	2.24
10^{-35}	Loop	1.06	4.04	4.21	0.0933	1.31	0.552	0.460
	Tree	1.52	2.98	6.52	0.109	1.67	0.518	0.260

Varying:	Density				Problem		
Fixed:	Butterfly, 1 soln				Symmetry, dense		
	10^{-5}	10^{-21}	10^{-35}	empty	1 soln	128 solns	256 solns
Loop	0.73	33.7	541.5	1.77	0.32	1.31	2.32
Tree	2.33	3.76	8.00	1.49	1.32	1.67	1.80

- for a fixed machine, input density, and problem, we may need the best parallelization as we vary the number of processors; and
- we may need the best parallelization for a fixed input density and problem as we port the program across machines.

One might assume that for some of these cases, the best parallelization does not vary, making the decision easy. In fact, we show in a detailed study of this application [Crowl *et al.*, 1994] that in *none* of these cases is the best parallelization fixed; the choice of which parallelization of subgraph isomorphism performs best varies in all cases by significant margins. Other researchers have also noted that the best parallelization for a given problem can vary depending on the input, machine, or problem [Eager and Zahorjan, 1993; Rao and Kumar, 1989; Subhlok *et al.*, 1993].

Examples of these effects are shown in Table 3.1. This table shows the best running time in seconds for loop- and tree-parallel implementations, while varying one component of the environment. The underlined entries in the table are the better-performing executions. The table shows that: 1) when we seek 128 solutions in a sparse solution space, some machines prefer loop parallelism, while others prefer tree parallelism; 2) when we seek 1 solution on the Butterfly, as the density of the solution space varies from 10^{-5} (dense) to 10^{-35} (sparse) and finally to an empty solution space, the best parallelization varies; and 3) in searching a dense solution space on the Symmetry, loop parallelism is preferable when seeking 1 or 128 solutions, but tree parallelism is preferable when seeking 256 solutions.

An example of how the best parallelization varies as we vary the number of processors used is shown in Figure 3.1. This figure shows the running time of four

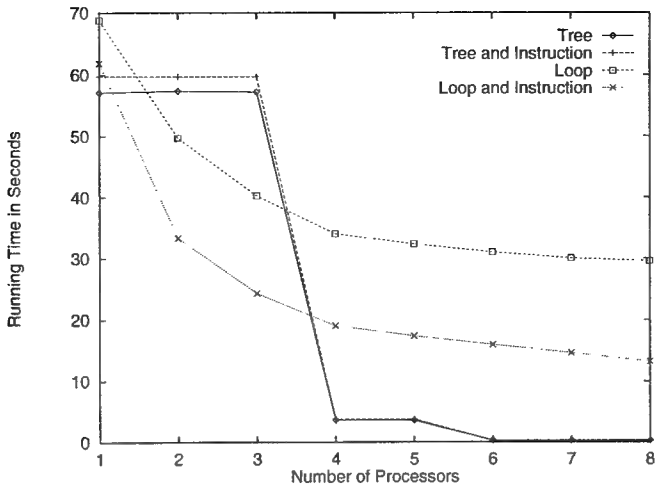


Figure 3.1: Comparison of Four Parallelizations, Varying Processors

parallelizations (tree, tree combined with instruction, loop, and loop combined with instruction) on the SGI. It shows that for some numbers of processors (1, and 4-8), tree parallelism outperforms the others; for other ranges of processors (2-3) loop and instruction parallelism outperforms the others; and neither loop parallelism nor tree combined with instruction parallelism ever perform better than all the others. It also shows that adding instruction-level parallelism to loop parallelism improves it significantly, while adding instruction-level parallelism to tree parallelism has a slightly negative effect on performance.

These effects can be explained in terms of load imbalance, communication costs, processor speed, pure computation costs, and speculative computation (parallelism used to discover alternative, cheaper solutions rather than speeding the discovery of a particular solution). Detailed explanations are presented in [Crowl *et al.*, 1994]; here we note only that the insight necessary to explain the relative performance of these implementations can be derived almost exclusively from high-level notions such as load imbalance and communication costs, without making detailed measurements of each execution.

We will use subgraph isomorphism as an example application in the following chapters because it demonstrates performance sensitivity to such a large number of external factors:

1. the number of processors used to run the program;
2. the structure of the input data set;
3. the problem definition; and
4. the machine used to run the program.

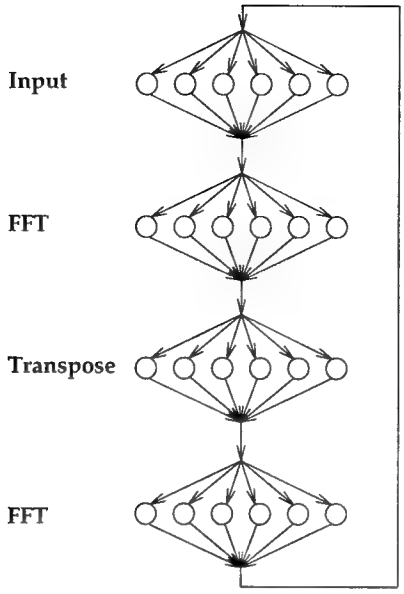


Figure 3.2: Data Parallel 2D FFT on 6 Processors

The wide performance sensitivity of subgraph isomorphism, combined with the large number of possible parallelizations of the application, make it an important test case for the utility of our performance measurement and prediction tools.

3.2 2D FFT

Our second example application is the two-dimensional discrete Fourier transform program (2D FFT). The serial implementation of the program consists of a number of iterations of 1D FFTs on columns of the input matrix, followed by 1D FFTs on the rows of the matrix.

We consider two parallel implementations of the program. The first parallelization is purely data parallel (DP). In the DP version, each iteration of the program consists of 5 parallel loops: one to initialize the matrix, one to perform the column-wise FFTs, two to transpose the matrix (using an intermediate matrix), and one to perform the row-wise FFTs. A schematic diagram of the DP version running on 6 processors is shown in Figure 3.2. The figure shows that all processors participate in each of the parallel loops in succession.

The second parallel implementation uses task parallelism as well as data parallelism. In this implementation (TP), processors are segregated into two groups using tasking directives. One group initializes the matrix and performs data-parallel row-wise 1D FFTs, while the other group transposes the matrix and

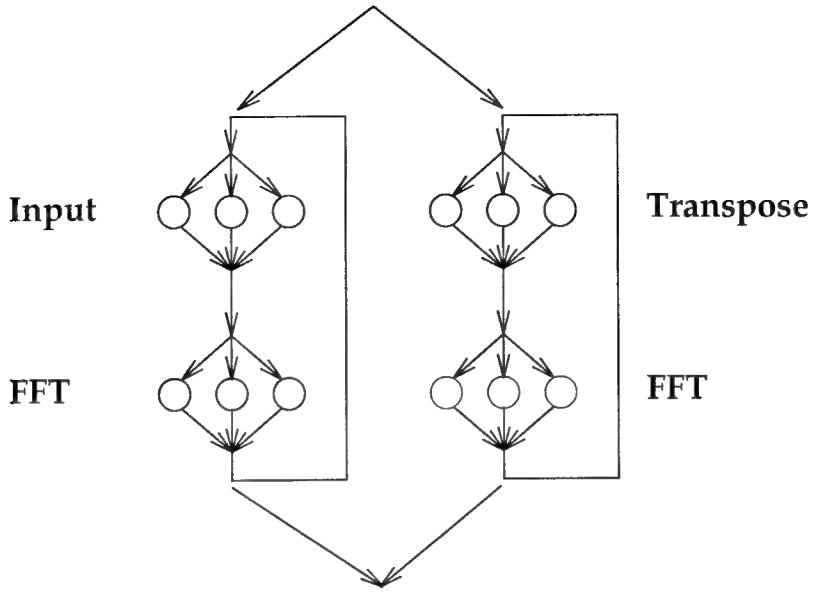


Figure 3.3: Task Parallel 2D FFT on 6 Processors

performs data-parallel column-wise 1D FFTs. The two tasks are pipelined so that each one is kept busy working on separate matrices. The TP version running on 6 processors is shown in Figure 3.3, in which 3 processors are assigned to the input-FFT task, and 3 processors are assigned to the transpose-FFT task.

A comparison of the task parallel and data parallel implementations of 2D FFT on the iWarp was presented in [Subhlok *et al.*, 1993]. On that machine, the authors discovered that as the data set size is varied past a certain threshold, the choice of which implementation is best changes. For small data sets ($n \leq 128$) the parallel tasking implementation outperformed the pure data parallel implementation. For large data sets ($n \geq 256$), the purely data parallel implementation outperformed the parallel tasking implementation. The principal reason for this effect is that in the parallel task version, communication between tasks must pass through a single channel of the iWarp network, while purely data parallel communication can take place along multiple channels. For small data sets, the larger problem granularity of parallel tasking leads to better performance, but as the problem size increases, intertask communication becomes a bottleneck.

It is interesting to ask whether a similar effect would be observed when this application is run on the KSR1, a machine with a significantly different architecture. Unfortunately, the data from the iWarp cannot help us decide which executions to measure, since the machines are so different. Thus we immediately run into a problem: perhaps there is a crossover between implementations for some external factors (here, n and p), but finding the crossover would require measurements over

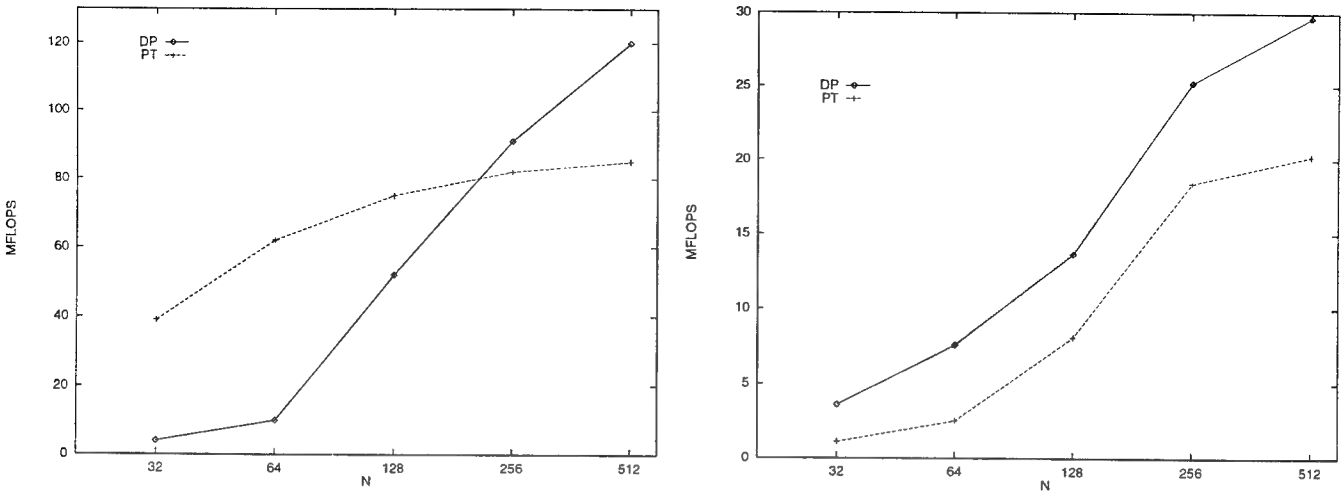


Figure 3.4: Comparison of Task and Data Parallel 2D FFT on the iWarp (left) and KSR1 (right).

the entire space.

In fact, we find that while the crossover exists on the iWarp, it does not occur on the KSR1, as is shown in Figure 3.4. This figure shows the performance in megaflops of the data parallel and task parallel implementations on the iWarp and on the KSR1. It shows the crossover that occurs on the iWarp, and data showing no crossover on the KSR1. We analyze this application in Chapter 5 to show why no crossover should be expected in general on the KSR1; the data shown here is intended merely to illustrate that performance tradeoffs between implementations that exist on one parallel machine may not exist on another machine.

The alternative implementations of 2D FFT provide a performance tradeoff worthy of study, since the choice of best implementation depends on the machine used, the number of processors used, and the size of the input dataset. In subsequent chapters we use 2D FFT as a primary test case for our performance prediction methods, because of the performance tradeoff it presents, and because its regular structure allows for analysis that is simple enough for use in examples.

4 Measuring Parallel Programs

The example problems discussed in the last chapter present difficulties to parallel programmers in two ways: first, in *understanding* the performance of a given execution; and second, in *predicting* the performance of a potential execution. In this chapter, we describe how to measure parallel programs in a way that leads to understanding for problem diagnosis and simultaneously is useful for performance prediction, using a method we call *predicate profiling*.

Predicate profiling is useful both for understanding and for prediction because it is based on *decomposition of overheads*, which helps in achieving both goals. Decomposition of overheads starts from the observation that the standard formula for parallel running time (discussed in Section 2.2.1)

$$T_P(n,p) = \frac{T_o(n,p) + T_c(n)}{p},$$

although most commonly used in scalability analysis, also has value from the standpoint of performance debugging. That is, perhaps the first question asked in performance tuning an application might be, "How much parallel overhead is present when executing this application?" The scalability analysis formula answers this by relating observed running time to parallel overhead.

Predicate profiling takes the decomposition process a step further and breaks T_o down into component parts, which we call *overhead categories*. We show in this chapter that the decomposition process can be done in such a way that the overhead categories offer significant utility to parallel programmers in attempting to understand an application. We will then show in the following two chapters how the same overhead categories can be used to provide the basis for performance prediction of parallel programs.

4.1 Decomposing Parallel Overhead

The use of parallelism in a program presents many new opportunities for performance degradation. Most parallel programmers are aware of these new sources

of poor performance, and group them into general classes, such as load imbalance, communication overhead, and synchronization loss. The use of these classes allows programmers to reason about and discuss the performance of specific programs. Unfortunately, as discussed in Section 2.1.3, few performance measurement tools directly support programmers in assessing the amount of performance degradation attributable to each of these classes.

Our decomposition-based approach to overhead measurement is based on a number of observations regarding the performance tuning process: programmers can identify categories of poor performance; poor performance is the result of the program or machine spending time in particular, inefficient, states; those states can be recognized, and time spent in them can be measured; and those states can be associated with categories of poor performance. Based on these observations, predicate profiling uses pre-defined expressions that recognize categories of poor performance and report specifically on *what is wrong* with a parallel program.

Predicate profiling attempts to provide a rapid, complete, assessment of a parallel program's performance in terms of categories that are semantically significant to programmers. This assessment either directly identifies opportunities for performance tuning, or serves as a focusing mechanism prior to the use of more detailed performance tuning tools. In order to accomplish this assessment, we suggest a candidate set of categories, and provide precise definitions for each category. We show that these definitions can be expressed in terms of functions that identify instantaneous program states; we call these functions *performance predicates*. An important feature of performance predicates is that, since they are formally defined, they allow precise measurements of program states that are often only informally defined, such as load imbalance. In this role, performance predicates form an unambiguous language for discussing the performance of parallel programs.

A performance predicate is a recognizer; it recognizes a state of the program that the programmer wishes to measure. In general, it can be very expensive to identify the instant when an arbitrary predicate changes value, especially in a parallel or distributed system [Kaelbling and Ogle, 1990]. Therefore, exact measurement of program states based on measuring each duration during which a predicate has a certain value is unattractive. Luckily, however, highly precise measurements are not required for performance debugging; we only need to know about any state that, in the aggregate, consumes more than some small fraction of running time. As a result, we can use a simple, sampling-based method that periodically evaluates each predicate, which we call *predicate profiling.*

We use the term *performance* predicate to distinguish performance predicates from correctness predicates. A correctness predicate must be evaluated atomically, which presents problems in parallel and distributed systems. A performance predicate on the other hand, need not *always* have a consistent interpretation when it

is evaluated. Simple performance predicates will be inconsistent only very rarely; with sufficiently frequent sampling, any errors introduced during evaluation will have negligible effect on the end results.

4.1.1 Properties of Performance Predicates

The proper choice of a set of performance predicates (and the states they recognize) is constrained in three ways:

Meaning. The states should represent categories of poor performance that are intuitively meaningful. There are three advantages in using meaningful states. The first advantage is that the resulting performance measurements are easily related to design decisions at the application level. For example, a predicate that measured "idle time" without distinguishing between that caused by load imbalance and that caused by serial code would aggregate measurements for different effects in a confusing way. The second advantage to meaningful states is that language independence is increased. Basing state definition on concepts that have significance to a programmer eases the implementation process for a new programming language. The third advantage is that states defined in a meaningful way tend to be easier to analyze mathematically, which will be discussed in more detail in the next chapter.

Completeness. The states should be complete. That is, they should include every source of performance loss. Completeness can be verified empirically: if, after measuring all overhead in a multiprocessor execution, the remaining computation equals that of the uniprocessor case, then the set is complete for that execution.

Mutual Exclusion. The states should be mutually exclusive. Mutual exclusion of states makes their measurements orthogonal — no wasted time is ever charged to two categories. This ensures that we can add and subtract overheads accurately, and makes it possible to reason about overheads independently.

Predicates chosen according to these constraints have the property that they express overheads in common units. That is, we can directly compare the overhead measured using one predicate to that measured using another, since they both recognize states in which no useful work is done. We will call time spent in these states in general *lost cycles* (LC). We usually express LC as its sum over all processors in an execution; this sum is equivalent to the value T_o. Although overheads are not independent (so we cannot in general vary one without changing another) the effect of changing LC in an execution is the same no matter

$$\texttt{Load\_Imbalance}(x) \equiv \texttt{Work\_Exists} \wedge \texttt{Processors\_Idle}(x)$$
$$\texttt{Insufficient\_Parallelism}(x) \equiv \neg\ \texttt{Work\_Exists} \wedge \texttt{Processors\_Idle}(x)$$
$$\texttt{Synchronization\_Loss}(x) \equiv \texttt{Work\_Exists} \wedge \texttt{Processors\_Spinning}(x)$$
$$\texttt{Braking\_Loss}(x) \equiv \texttt{Solution\_Found} \wedge \texttt{Processors\_Busy}(x)$$
$$\texttt{Memory\_Loss}(x) \equiv \texttt{Processors\_Stalled}(x)$$
$$\texttt{Wasted\_Computation}(x) \equiv f(\texttt{Events}, \texttt{States})$$

Figure 4.1: Candidate Set of Performance Predicates

what specific kind of overhead is actually changing. Thus, LC forms a single uniform metric for comparing different overheads that are defined by performance predicates.

In summary, the set of performance predicates should form a meaningful, orthogonal basis set for the space of performance overheads; the norm we use in this space is the metric LC. Although it may not be obvious that such a set exists, the next section presents a set that meets these criteria; the remainder of the paper discusses the use of this set and demonstrates its utility.

4.1.2 Defining a Set of Performance Predicates

The set of performance predicates we use in our initial examples are defined in Figure 4.1.[1] Most predicates are defined based on global or per-processor states, which are the interpretations of the expressions on the right hand side of the equations in the figure. For example, the first predicate is read as "A load imbalance on x processors exists if `Work_Exists` is true and `Idle` is true for exactly x processors."

If processors are idle, their processing power is being wasted; the first two predicates express this, and distinguish between load imbalance and insufficient parallelism as causes of idling. If processors are spinning (defined to be true while waiting for synchronization), then their processing power is being wasted only if work exists to be done; the third predicate captures this case. If processors are busy, their work is being wasted only if the solution has already been found; the fourth predicate expresses this case. The fifth predicate (`Memory_Loss`) recognizes inaction while waiting for service from memory. This predicate thus includes communication costs in a shared-memory multiprocessor as well as startup and replacement misses.

The last predicate (`Wasted_Computation`) is defined by the user; it expresses algorithmically wasted computation. Wasted computation occurs in some pro-

[1] The names of some of these predicates are taken from [Møller-Nielsen and Staunstrup, 1987].

grams because adding processors changes the actual work done by the algorithm [Lai and Sahni, 1984]. The work added may not contribute to solving the problem; providing this category allows the programmer to interpret and treat this wasted work as a form of overhead.

The following section demonstrates how the use of performance predicates imposes an easily-understood and easily-implemented structure on the performance debugging and tuning process. In addition, our examples show that performance predicates provide a common basis for the comparison of different kinds of overhead in a parallel program. This common basis is especially useful in program design and application-level tuning, and can be used to decide when a program is performing well when other metrics (*e.g.*, speedup) are ineffective.

4.2 Predicate Profiling

The precise definitions of the performance predicates in the last section provide a basis for quantitative evaluation via predicate profiling. In this section we describe two implementations of predicate profiling. The first version is implemented for the Silicon Graphics Challenge Series multiprocessor (*SGI*) and uses a sampling-based approach. We describe this implementation in detail to explain predicate profiling. The second version is implemented for the Kendall Square Research KSR1 multiprocessor, and is based on event-logging. We describe the KSR version only in terms of its differences from the SGI implementation.

4.2.1 SGI Implementation

Sampling-based predicate profiling could be done by system software on each processor, but for the SGI implementation we allocated one of the system's processors to the sampling task, as in Parasight [Aral and Gertner, 1988].

An advantage of using predicate definitions that are primarily posed in terms of global and per-processor states is that they can be implemented by setting flag-variables in a shared-memory environment. This allows profiling to be implemented by a process that periodically inspects shared memory.

Our global states are `Work_Exists` and `Solution_Found`. `Work_Exists` is true whenever parallel work has been created, but not yet completed; for example, between a **fork** and its corresponding **join**. `Solution_Found` is true when the program has completed its necessary computation; unnecessary computation may still be occurring. That is, `Solution_Found` is true as soon as the program *could* print out the answer it is intended to compute.

Our per-processor states are `Busy`, `Spinning`, and `Idle`. `Busy` is true for a processor when it is executing code that is logically a part of the program. For

example, `Busy` is true when a processor begins executing its iterations of a parallel loop, and if the processor does not execute the subsequent serial code, `Busy` is false when its loop iterations complete. `Spinning` is true for a processor after it has requested a lock, but before it has acquired the lock (`Spinning` overrides `Busy`). `Idle` is true for a processor whenever it is not `Busy` or `Spinning`.

These flag definitions are intuitive, which is an essential feature. Because of their unambiguous definitions, decisions on where to set and clear such flags in the source are very straightforward, which allows accurate implementation in a wide variety of applications. In fact, we believe that these flag manipulations can be incorporated in self-instrumenting macros, such as the Argonne P4 macros, allowing predicate profiling to be totally hidden from the programmer.

Profiling the `Memory_Loss` predicate is more difficult. On most machines, processors stall while waiting for memory, so this predicate cannot be defined based on an observable processor state, since the processor cannot itself determine when it is stalled. The most common way this state is measured is via simulation, as in [Goldberg and Hennessy, 1993]; we use that method in the SGI implementation.[2]

Measuring the `Wasted_Computation` category is naturally application-dependent, and so it requires programmer definition; however, in our application it is easily identified. We define a state corresponding to each independently-schedulable piece of work (task), and if the processor, after executing the task, has not contributed to the solution it signals so via an event. The event causes the time measured for that task to be charged to `Wasted_Computation`. Since wasted computation may occur in many forms, this approach is not completely general, but the underlying mechanisms are sufficient for a wide variety of cases.

The cost of sampling-based predicate profiling in a shared-memory multiprocessor amounts to the additional communication generated by the sampling process (along with the use of the dedicated processor). This cost can be reduced by sampling at less than the maximum possible rate; however, lowering the sampling granularity increases the possibility of error in the results. Luckily, neither of these effects is very severe. Figure 4.2 shows both the percent error introduced in a profile and the percent increase in running time for a typical run of our example application. The figure shows that for a wide range of sampling intervals (between about 50 and 250 μs), both running time overhead and measurement error are close to about 5%.

[2]Although our final KSR implementation uses event logging, we have been able to profile the `Memory_Loss` predicate at run-time using the on-line instruction counting implemented by compilers on the KSR1 [Kendall Square Research, 1991]. KSR compilers update a dedicated register with the current instruction count for each basic block (at minimum). Usually this can be done by replacing no-ops in the instruction pipeline, so the cost is minimal. Measuring this predicate requires that we (automatically) modify a program's assembly code so that each basic block stores the current instruction count to local memory — this makes the stall behavior of each processor visible at run time.

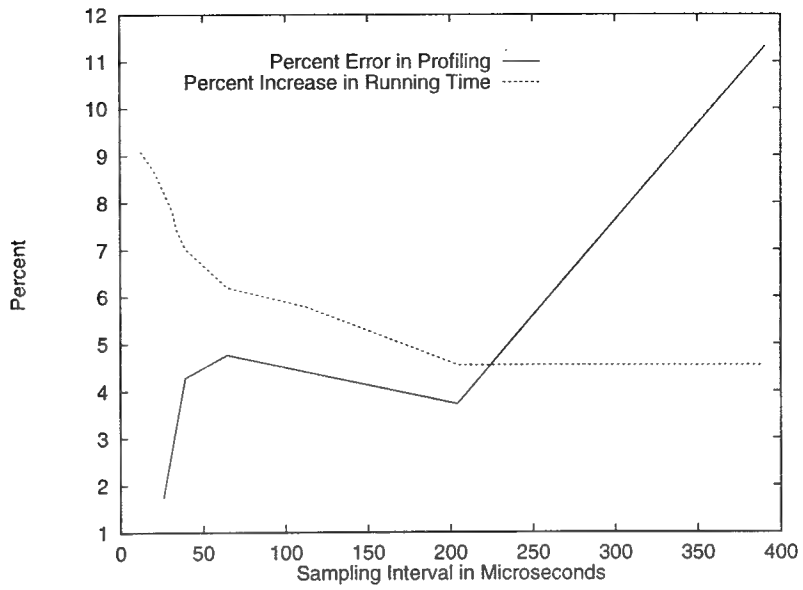


Figure 4.2: Measurement Error and Running Time Increase For a Range of Sampling Intervals

The effort involved in predicate profiling is relatively small. The sampling task is structured as 50 to 100 lines of code (depending on the number of predicates) which periodically evaluate each predicate. The flag-setting code amounted to adding only 18 lines of code to our application. The code for the sampling task is reusable across implementations, and the flag setting code can be made invisible by embedding it in the runtime system.

The results of applying predicate profiling to an application can be presented in a number of ways. We have found that the simplest presentation is often the most useful: a summary table for each execution of the program. The table shows the total time, summed over all processors, that the program spent in each category. It provides an immediate assessment for the programmer of how the program performed, and what sort of tuning would be profitable. It allows the programmer to make an informed tradeoff between different kinds of tuning ($\epsilon.g.$, memory system tuning vs. load balance tuning) and the relative effort that each kind of tuning requires.

Other useful presentations expand the profiling data along one of a number of dimensions: processors, time, or code. In Section 4.3 we show an example where per-processor profiles give insight to performance problems. We have also expanded the profiling data in the time dimension, and can present the results using the display tool *Upshot*, available from Argonne National Labs. Finally, we believe that by combining predicate profiling with traditional program-counter profiling, overhead values could be associated with procedures or loops, allowing more detailed exploration of code segments during performance tuning.

4.2.2 KSR Implementation

The KSR implementation of the predicate profiler, measures Fortran programs running on the Kendall Square KSR1, and consists of 1) a library linked into the executable code (`pplib`) and 2) a post-processor of event logs that outputs the profile (`pp`). The KSR Fortran runtime system can log events such as the start and end of individual loop iterations, which we use for calculating load imbalance. Additional calls to our library routines are inserted at the start and end of parallel loops, parallel tasks, and synchronization operations. The inserted library calls are quite simple and could easily be added by a source-to-source preprocessor. Manual pages describing the user interface to `pp` and `pplib` are presented in Appendix A.

The category set we use in the KSR implementation is:

Load Imbalance: processor cycles spent idling, while unfinished parallel work exists.

Insufficient Parallelism: processor cycles spent idling, while no unfinished parallel work exists.

Synchronization Loss: processor cycles spent acquiring a lock, or waiting in a barrier.

Communication Loss: processor cycles spent waiting while data moves through the system.

Resource Contention: processor cycles spent waiting for access to a shared hardware resource.

We revised our category set for the second implementation for a number of reasons. First, the categories of Braking Loss and Wasted Computation were less relevant to the regular, scientific computations which are our primary target applications on the KSR. Second, the distinction between Memory Loss and Resource Contention is important for performance modeling; these overhead categories tend to exhibit significantly different behavior.

This category set has satisfied the three criteria (completeness, mutual exclusion, and meaning) for the applications we have studied, which are numerically-oriented, scientific computations. Of course, it will need to be expanded to handle a wider range of overheads as it is used in more varied situations. In particular, it does not currently distinguish between synchronization types, measure contention for software resources, or measure operating system and runtime library effects. Each of these extensions appears to be straightforward within the existing framework however.

The KSR1 [Kendall Square Research, 1991] is a two-level ring architecture in which all memory is managed as a cache, which is organized in two levels on each node. Thus, inter-node communication occurs only as the result of misses in the secondary cache. Dedicated hardware monitors the state of buses between the processor and the second-level cache. This performance monitor counts the number of secondary cache misses, the time taken to service secondary cache misses, and the number of cache lines that pass through the higher-level ring before arrival. Based on this data, we can calculate the amount of communication performed in an execution and the amount of resource contention that occurred.

Load Imbalance, Insufficient Parallelism, and Synchronization Loss are defined and measured as in the SGI implementation. Communication loss is measured as a simple product of the number of cache misses and the ideal time to perform the cache line transfers. Resource contention (contention for the ring interconnect and for remote memories) is measured as in [Tsuei and Vernon, 1990] — that is, the ideal time to perform the communication operations is compared to the actual elapsed time. Since the KSR1 hardware monitors both the number of cache lines transferred and the elapsed time waiting for cache lines, this calculation is straightforward. Although the performance monitoring hardware on the KSR is rather unique, something comparable may be required for other cache-coherent architectures. On simpler architectures, such as a message-passing system, the performance monitoring capabilities of the DEC Alpha [Digital Equipment Corporation, 1992] should be sufficient to gather the same information.

pp is currently installed for use by the user community at the Cornell Theory Center on their 128-node KSR1. Example output from the current version of pp is shown in Figure 4.3. Lost cycles for each category are presented in seconds, aggregated over all processors. Actual execution time of the application was 9.86 seconds, which is equal to total time (49.29 seconds) divided by the number of processors (5). In this execution, pp has identified that the primary bottleneck is a serial section of code.

4.3 Predicate Profiling in Practice

We have found predicate profiling to be useful in application-level performance tuning (*e.g.*, selecting the best parallelization of an algorithm) and in program-level tuning (*e.g.*, improving one particular parallelization of a program). The next two sections use as their example problem parallel subgraph isomorphism (described in Section 3.1). Section 4.3.1 gives three examples of application-level tuning, and section 4.3.2 gives an example of program-level performance tuning.

As discussed in Section 3.1, there are many opportunities for parallelism available in subgraph isomorphism, including tree parallelism, loop parallelism, instruction parallelism, vector operations and functional parallelism. Additionally,

```
% f77 -o runfast -r8 -para myprog.f -lctc -lpmon
% runfast
% pp
  PP version 4.0
  ** processors: 5
  Load Imbalance            0.180677
  Insuff Parallelism       16.813304
  Synchronization Loss      0.003779
  Communication Loss        2.274899
  Resource Contention       1.351362
  Total Time               49.293890
  Remaining Time           28.669868
```

Figure 4.3: Example Output of pp

we might choose to parallelize only certain loops, or certain subtrees. As a result, there is a very large number of possible parallelizations of this problem. Without an understanding of the reasons why one parallelization outperforms another for a particular machine, choosing the best parallelization for this algorithm would be very difficult. Most performance evaluation tools provide too much low-level detail to quickly assess each of these parallelizations; in contrast, we show in the next section how predicate profiling can quickly and simply explain the important factors determining the best parallelization for subgraph isomorphism on a given machine.

4.3.1 Choosing the Best Parallelization

Predicate profiling seems especially well suited to the design and application-level tuning of parallel programs. In this section we present examples that show how predicate profiling can explain why:

1. different parallelizations of a program may exhibit widely differing performance;

2. different parallelizations of a program may exhibit similar performance for different reasons; and

3. the effect on performance of porting to a new machine could be very different for different parallelizations.

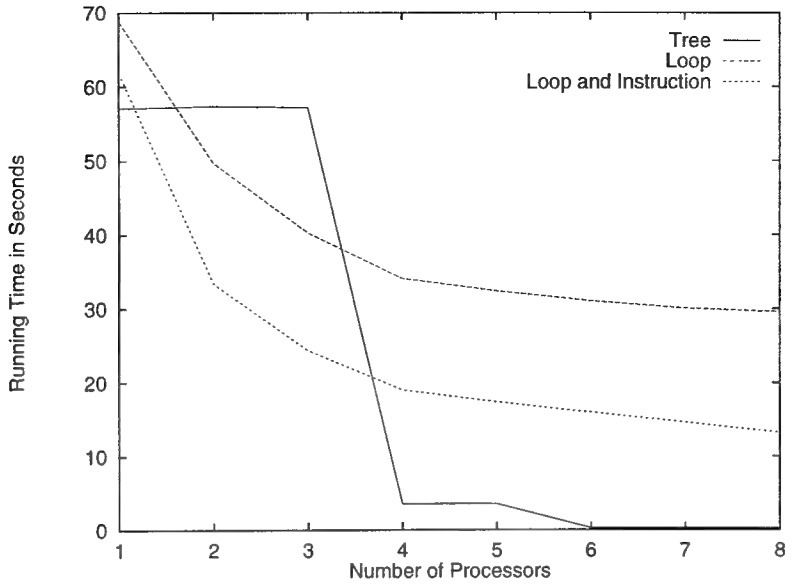


Figure 4.4: Running Time of Three Parallelizations on SGI; One Solution, Sparse Input

Our first example examines the reasons for the widely differing performance of three parallelizations of subgraph isomorphism. We are concerned with the best parallelization when searching for a single isomorphism, given a sparse input space. The running times of three parallelizations: tree, loop, and loop plus instruction, are shown in Figure 4.4 for the SGI. This figure shows that there are a number of issues involved in understanding the performance of these parallelizations: why is tree parallelism the worst performer on 2 and 3 processors, but the best performer on 4 or more processors? Why is loop plus instruction parallelism better than loop alone? What is preventing loop parallelism from performing well? Will tree parallelism always outperform loop plus instruction on more than seven processors?

Figure 4.5 shows predicate profiles for the three executions shown in Figure 4.4. In these charts, each bar shows the total processing time, summed over all processors. Each bar is further broken down to show computation time (lowest segment) and the three kinds of LC that are significant in this case: Load Imbalance, Wasted Speculation, and Memory Loss.

The leftmost profile shows the overheads in tree parallelism. On 1 through 3 processors, we see that LC due to wasted speculation entirely accounts for the lack of any speedup. The increases in wasted speculation imply that, in this sparse solution space, the branches searched by processors 2 and 3 do not yield a solution before the branch searched by processor 1. However, the fourth processor, exploring its own branch, finds a solution much sooner than does processor 1; likewise, processor 6 also improves on processor 4's solution time. Thus this

figure gives us good insight into the distribution and relative cost of solutions in the search tree, and explains why tree parallelism outperforms loop and loop plus instruction for processors greater than 3.

Next, we examine the other two profiles. The loop profile (in the center) shows that the LC constraining this parallelization is both communication and load imbalance. Comparing the loop profile to the loop plus instruction profile, we see that computation cost is actually higher when using instruction parallelism — instruction parallelism itself is not outweighing the costs of packing and unpacking data, which must occasionally take place under instruction parallelism. However, the profiles show a less-expected benefit of this parallelization: the smaller dataset size created by packing data leads to lower communication costs when using instruction parallelism. The advantage of the common use of LC for comparing overheads in this example is that we can directly observe how instruction parallelism improves execution time: the increase in computation costs caused by data manipulations are more than offset by the decrease in LC attributable to communication. Since there is little communication-caused LC in the tree parallel execution, we expect instruction parallelism to have little benefit in combination with tree parallelism. Our data confirm this conclusion: instruction parallelism for this problem *increases* the running time of tree parallelism anywhere from 5 to 25%.

The effects of speculative parallelism seen in the tree parallel version demonstrate that for some programs, it can be difficult to determine when they are performing well. Speedup is not an effective metric for this program because the multiprocessor execution does not compute the same result as the uniprocessor execution. However, LC can be used to assess whether the program is performing well, since the absence of LC, along with an efficient uniprocessor algorithm, indicates an efficient program.

Our second example shows two parallelizations that exhibit different reasons for good performance. Figure 4.6 shows the running time of instruction plus loop, and instruction plus tree parallelism when searching for many solutions in a dense solution space.

The similar performance of these two versions occurs for different reasons, as the profiles in Figure 4.7 show. Both parallelizations start out finding the same set of solutions, as can be seen by their comparable profiles in the one processor case. However, as we add processors, tree parallelism (on the left) benefits from finding some solutions faster in each additional subtree. This benefit is shown by the fact that each additional processor adds some pure computation, but not as much as the 1 processor case. On the other hand, loop parallelism (on the right) still finds the same solutions, only faster. It is in contrast being limited by load imbalance and memory loss (communication) as the number of processors increases.

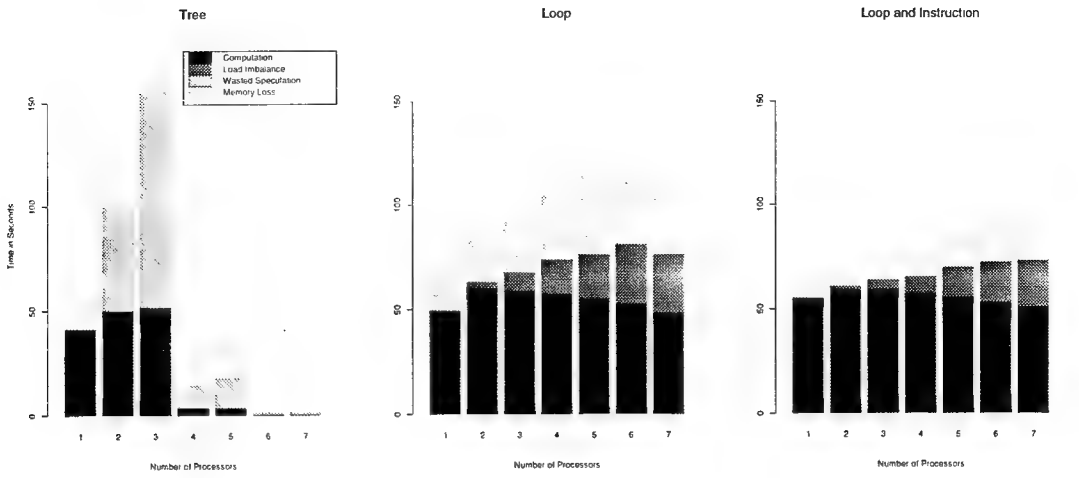


Figure 4.5: Predicate Profiles of Three Parallelizations on SGI; One Solution, Sparse Input

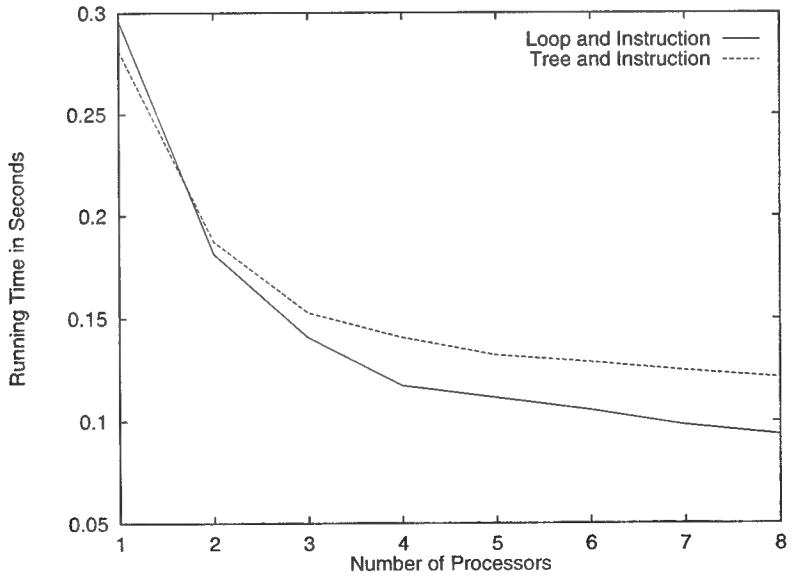


Figure 4.6: Running Time of Two Parallelizations on SGI; Many Solutions, Dense Input

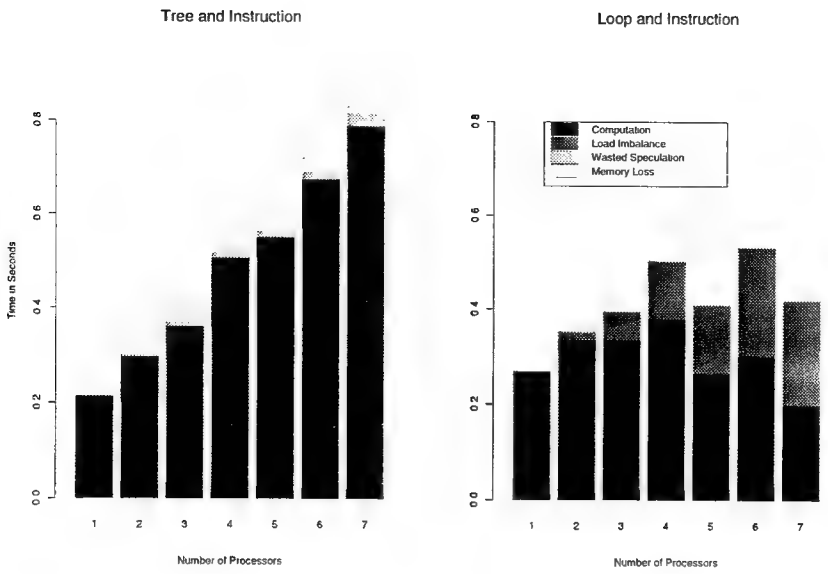


Figure 4.7: Predicate Profiles of Two Parallelizations on SGI; Many Solutions, Dense Input

By using LC as a common metric, these data show that the two parallelizations are exploiting entirely different sources of performance improvement on the same problem. The predicate profiles indicate that if we are interested in porting this application to a machine with a higher communication-to-computation ratio, we would then prefer tree parallelism for this case; conversely, on a machine with a lower communication-to-computation ratio we should use the loop parallel version. Finally, predicate profiling in this case suggests that there might be an opportunity for a hybrid algorithm that exploits both tree and loop parallelism, a fact which is confirmed in [Crowl *et al.*, 1994].

In our third example, we show how predicate profiling can be used to understand a case in which loop parallelism outperforms tree parallelism on the SGI, while tree parallelism outperforms loop parallelism on the KSR1. When searching for multiple solutions in a sparse input space, we find that the two parallelizations perform as shown in Table 4.1. This table shows that the proper choice of parallelization depends on the underlying machine.

To understand why the SGI outperforms the KSR under loop parallelism, we first note that the uniprocessor (sequential) running time of the program is 21.88 seconds on the KSR, while it is 8.66 seconds on the SGI. Although the SGI is faster at solving this problem on a single processor, the SGI only has 8 processors, while our KSR configuration has 32 processors (much larger machines are available). Our measurements of load imbalance show that for this problem, on these machines, the degree of load imbalance under loop parallelism grows quite large with an increase in the number of processors. Figure 4.8 shows the fraction

Table 4.1: Running Time of Loop and Tree (in seconds); Multiple Solutions, Sparse Input

	SGI	KSR1
loop	2.05	10.68
tree	2.59	2.24

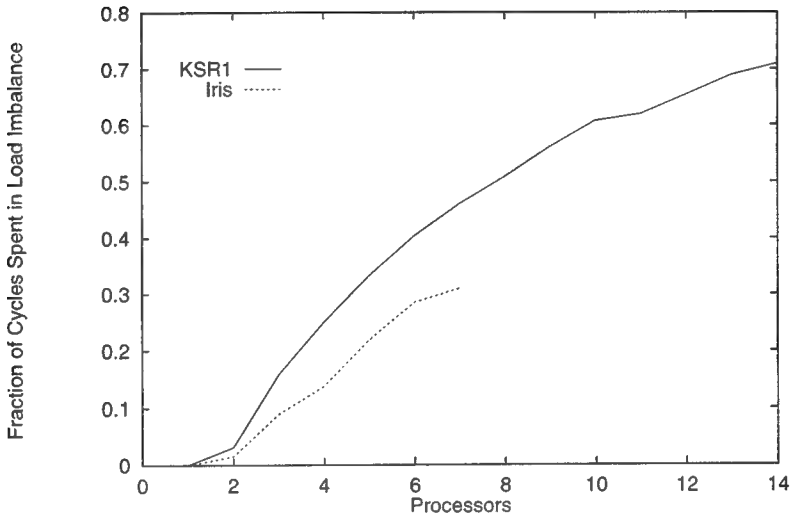


Figure 4.8: Increasing Load Imbalance in Loop Parallelism

of total processor cycles lost due to load imbalance for loop parallelism on this problem on both machines. The figure indicates that beyond about 8 processors, the fraction of cycles lost due to load imbalance grows very large. In fact, the benefit of adding additional processors beyond this point is completely counteracted by the increase in load imbalance, precluding the KSR from benefiting from its larger supply of processors.

On the other hand, the KSR outperforms the SGI under tree parallelism. As before, the single processor case favors the SGI (7.03 seconds on the SGI, 18.86 on the KSR1). However, there is no load imbalance under tree parallelism on this problem; the dominant source of LC is wasted computation due to speculation. Figure 4.9 shows the total time spent on wasted computation for this problem on both machines, in seconds. As the number of processors increases, each time the line does not rise, the program has benefited from an increase in processing power. When the line stays flat, a constant amount of work has been divided among a larger number of processors, and when the line drops, the program has

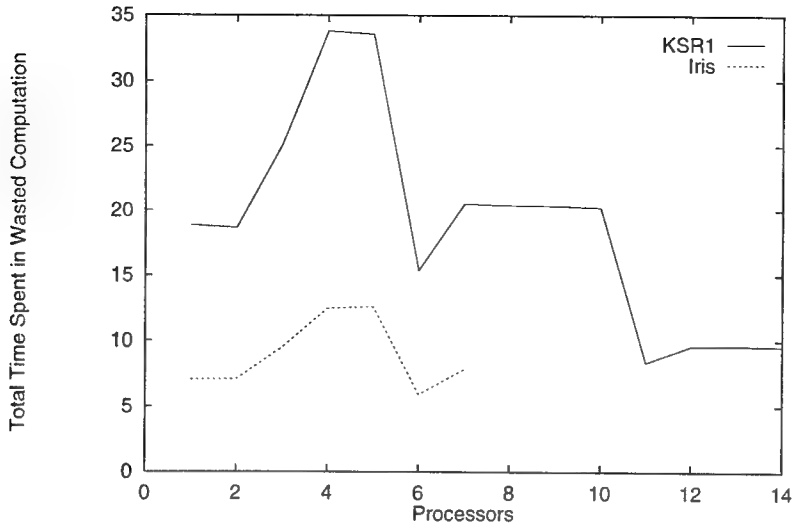


Figure 4.9: Decreasing Wasted Computation in Tree Parallelism

found a cheaper set of solutions via speculative parallelism. The figure shows that increasing processors for this problem continues to yield significant benefits beyond 8 processors. As a result, the KSR is able to exploit its larger number of processors to advantage and outperform the SGI. Using the common metric of LC in both cases allows us to confirm that the KSR should outperform the SGI for tree parallelism after about 11 processors, since at that point the KSR execution contains the same or less LC than does the 8-processor SGI execution.

In this case we have seen that the LC due to speculation decreases with an increase in processors, while the LC due to load imbalance increases with an increase in processors. The tradeoff between these two sources of LC indicates that tree parallelism is favored on machines such as the KSR which can provide many processors, while loop parallelism is favored on machines such as the SGI which have a smaller number of faster processors.

4.3.2 Tuning A Parallelization

While the previous section concentrated on the use of predicate profiling in the design or application-level tuning of a program. in this section we show an example of how predicate profiling can help in tuning an application whose parallel structure has been determined.

We have seen that load imbalance is one of the principal contributors to poor performance in subgraph isomorphism. We will now study the last example in the previous section in more detail, to see if predicate profiling can offer insight that

Table 4.2: Per-processor Load Imbalance; Multiple Solutions, Sparse Input, KSR1

Processor ID	0	1	2	3	4	5	6	7
Load Imbalance	4.54	5.13	5.05	4.96	4.32	4.03	4.09	3.54

could lead to removing some load imbalance. To do so, we separate out the load imbalance measurements on a per-processor basis. The resulting measurements for 8 processors on the KSR1 (the program showed minimum running time in this case) are given in Table 4.2.

Table 4.2 indicates that there is a systematic source of load imbalance present in the parallel loops, which is indicated by the steadily decreasing values for load imbalance with processor number (after excluding processor 0, which is a special case). In this program, loop iterations are statically scheduled in a blocked fashion, with processor 1 getting the first I/N iterations, processor 2 getting the next I/N, and so on (where I is the total number of iterations in the loop, and N is the number of processors participating). The load imbalance values indicate that loop iterations with lower-numbered indices tend to have less work to do. The reason this occurs is that each iteration of a filter loop corresponds to eliminating search nodes at one particular tree level. Later iterations correspond to search nodes closer to the leaf level. When the program searches downward to some level n, all levels closer to the root than n contain only one search node, representing the path taken to the current node. Tree levels closer to the leaves than n will in general contain many search nodes. Since the filters operate by trying to eliminate search nodes one by one, the filters do more processing in later loop iterations (in the average case).

The presence of systematic load imbalance suggests that a simple modification of the program, to a round-robin scheduling of loop iterations, might alleviate some load imbalance. In such a schedule, processor 1 is assigned iterations numbered $1, 1+N$, etc. In fact, this simple modification, suggested by the predicate profile, results in a performance improvement of 12%, decreasing running time from 10.22 seconds down to 9.04 seconds on 8 processors.

This sort of extension of predicate profiling can be thought of as an instance of the general notion of *predicate refinement*. Predicate refinement is the way in which predicates are made to apply to a more restricted domain, so as to narrow the scope of investigation and pinpoint performance problems.

For example, as written, the `Load_Imbalance` predicate does not distinguish between the two common kinds of load imbalance: unequal task sizes, and unequal numbers of tasks on per-processor work queues. It would be a simple and consistent extension of these predicates to make that distinction, by adding a per-

processor state that indicates an empty work queue. We would then refine the `Load_Imbalance` predicate into two disjoint predicates, based on the state of the per-processor work queue, providing better insight into the kinds of overhead being measured. Other predicate refinements include restricting predicate evaluation to certain segments of code, or differentiating among many code segments. Highly detailed predicate refinement would consist of sampling the program counter as well as the performance predicates, and relating overheads to code segments at the source line level.

4.4 Summary

In this chapter we have presented a performance measurement approach based on the notion of decomposing parallel overheads. We have argued that the important conditions for a useful decomposition of overheads are completeness, mutual exclusion, and meaningfulness. Two implementations of a tool the embodies our approach, called predicate profiling, were described and shown to be helpful in interpreting anomalies and tuning the performance of parallel programs.

Predicate profiles serve a number of roles. First, they provide insight into the root causes of performance degradation; they show quantitatively and meaningfully why an application is not achieving good speedup. Second, they serve as a precursor to more detailed analysis. Detailed performance analysis tools usually focus on a single type of overhead, so predicate profiles compare quantitatively the performance gain possible by tuning with different tools. Finally, predicate profiles can serve as a basis for performance prediction, as we will show in the next chapter.

5 Predicting The Effects of Varying the Execution Environment

The examples in the previous chapter focused on the use of predicate profiling as an aid to understanding the performance of parallel programs. In this chapter we describe the further use of predicate profiles as a basis for performance prediction. In this chapter, our goal shifts from understanding program performance, to predicting the performance of parallel programs.

Section 2.3 pointed out that techniques for performance prediction of parallel applications include static and dynamic components. Our approach to performance prediction is distinguished by the distinction it contains between its dynamic and static components. The dynamic component of our performance prediction approach, *lost cycles analysis*, is concerned exclusively with factors *external* to the application. The static component of our approach, *lost cycles modeling*, is likewise restricted to factors *internal* to the application; lost cycles modeling will be discussed in the next chapter.

As discussed in Chapter 3, external factors are the number of processors used to run the program, the size of the input data set, the structure of the input data set, the problem definition, and the machine used to run the program. Lost cycles analysis takes a dynamic approach to discovering performance models of the application as a function of each of these external factors. The ability of lost cycles analysis to address external factors using only runtime measurements is based on the leverage obtained from overhead decomposition. Decomposing overheads not only leads to greater value as a tool for understanding, but exposes many typical behaviors that can be exploited to quickly develop performance models.

This chapter describes how lost cycles analysis uses overhead decomposition to assist the user in developing a performance model of an application. The following three sections 1) present overhead decomposition and show why it simplifies the program modeling process; 2) describe how lost cycles analysis is performed in practice; and 3) present case studies showing the effectiveness of lost cycles analysis.

5.1 Modeling Parallel Overhead

In this section we describe the methods we use to model parallel overhead and the tools we have developed to support the modeling process.

5.1.1 Methods of Modeling Parallel Overhead

To perform lost cycles analysis, we again start from the scalability analysis equation

$$T_P(n,p) = \frac{T_o(n,p) + T_c(n)}{p}.$$

This equation shows that we can predict the performance of a parallel application based on the external factors n and p if we know how T_o and T_c vary as functions of n and p. Lost cycles analysis extends the domain of T_P, T_c, and T_o to include all external factors, whose state we will denote by E. We assume that each external factor can be expressed as a real number; for example, the input data structure in subgraph isomorphism can be expressed in terms of the density of the solution space. Thus lost cycles analysis uses as its fundamental equation

$$T_P(E) = \frac{T_o(E) + T_c(E-p)}{p}$$

where $E - p$ denotes the state of all external variables except p.

Lost cycles analysis is the rapid formulation of empirical models for $T_o(E)$ and $T_c(E-p)$. The function $T_c(E-p)$, which represents the pure computation in the application, can be obtained using only uniprocessor executions. The process of finding $T_c(E-p)$, while not concerned with issues of parallelism, is supported well by the approach we take to find $T_o(E)$. For this reason most of the following discussion will focus on how we find $T_o(E)$, with the understanding that similar (though simpler) methods can be used to find $T_c(E-p)$.

In the same way that we gained understanding of program behavior by decomposing parallel overhead measurements, we gain leverage in predicting program behavior by decomposing parallel overhead models. We will denote the overhead categories defined in the KSR implementation of predicate profiling as follows: Insufficient Parallelism – IP; Load Imbalance – LI; Synchronization Loss – SL; Communication Loss – CL; and Resource Contention – RC. Using these terms, we can express total parallel overhead as

$$T_o(E) = IP(E) + LI(E) + SL(E) + CL(E) + RC(E).$$

The advantage gained by this decomposition arises because, while the overall expression for $T_o(E)$ may be quite complex, each of the component expressions (*e.g.*,

Table 5.1: Typical Functional Forms for Overhead Categories

Category	Variable	Typical Form	Citation
LI	n	$k_1 n + k_2$	[Hummel *et al.*, 1992]
LI	p	$k_1 p\sqrt{p} + k_2$	[Zhang and Srinivasan, 1990]
IP	n	k_1	
IP	p	$k_1 p$	[Amdahl, 1967]
SL	p	$k_1 \log p + k_2$	[Markatos *et al.*, 1991]
CL	n	$k_1 n$	
CL	p	$k_1 p + k_2$	[LeBlanc, 1988]
RC	p	$\max(k_1 p + k_2, k_3)$	[Lazowska *et al.*, 1984]

$IP(E)$) can be much simpler. Discovering these simpler expressions from a direct examination of performance data is a feasible task for parallel programmers, especially when the process is highly automated by a tool like `lca`, which incorporates *a priori* knowledge of many typical behaviors of the component expressions.

The typical behaviors of the various overhead categories are a small set of functional forms (formulae with unspecified constants), specific to each category. Examples of typical overhead behaviors as functions of n and p are shown in Table 5.1; for most behaviors the table also gives an example from the literature in which the behavior is documented. The table is representative and does not include all typical behaviors; the complete list is embodied in `lca`, currently with at least three typical behaviors per category.

As can be seen from Table 5.1, typical behaviors are expressed as functions of a single variable. Each typical behavior is therefore the *projection* of an overhead's performance function along a single axis. For example, the first entry in Table 5.1 states that a typical projection of $LI(E)$ along the n axis is $k_1 n + k_2$. Since E may have more than one element of interest (we may be interested in more than one external factor) a number of measurements must be made for each such element of E. However, the number of external factors of interest is typically small; for many applications we are only concerned with the external factors n and p.

5.1.2 Tools For Modeling Parallel Overhead

In order to perform lost cycles analysis, we must 1) measure actual executions to gather the necessary dynamic performance data; and 2) select the proper models for each overhead category. The next two sections describe our tool support for those tasks.

```
% pp -l "d=32"
d=32 v=4.5 p=2 li=0.018748 ip=0.002202 sl=0.276536
cl=0.077673 rc=0.037759 tt=1.313014 rt=0.900096
```

Figure 5.1: Example Output From `pp` Using The `-l` Flag

Using the Predicate Profiler in Lost Cycles Analysis

We use the KSR implementation of the predicate profiler, `pp` (described in Section 4.2.2) to gather performance data from the application. We have extended `pp` to provide support for this process by creating a new, concise output format better suited to the collection of data over multiple executions.

By specifying the `-l` flag, data is formatted in a manner for use by `lca`. In addition, any text placed after this flag is put into the execution log. The additional text can be used as an identification tag on the data from this execution indicating the value of all environment variables. An example of the use of `pp` with the `-l` flag is shown in Figure 5.1. Lines such as those shown in Figure 5.1 can be accumulated in a file, resulting in a complete record of the performance data necessary for `lca`.

A Model Selection Tool: `lca`

`lca` is a tool that manages performance data and focuses the user on a selection of appropriate models for each category of lost cycles. It assists the user in two ways:

1. It guides the user's selection of models for each category and environment variable, using defaults based on the typical behaviors of each category of lost cycles.

2. It provides error estimates for the goodness-of-fit for each of the default models, and for any models explicitly requested by the user. These estimates give the user the opportunity to compare the quality of each of the default models and to additionally compare any models the user feels might be better than the defaults.

The manual page detailing the user interface for `lca` is presented in Appendix A. An example output from `lca` is shown in Figure 5.2. This example first shows the collection of a number of predicate profiles for the program `runfast`, in which the number of processors used (p) is varied. The resulting datafile is then processed by `lca`. The arguments given to `lca` specify that we are interested in

```
% runfast -p 1
% pp -l >> runfast.pp
% runfast -p 2
% pp -l >> runfast.pp
% runfast -p 3
% pp -l >> runfast.pp
[...]

% lca -v p -c li -f runfast.pp
lca v.  2.0.
Datafile: runfast.pp, Variable: p, Category: li, Constraints: d=32
 (0.01022 +/- 0.00316) * p * sqrt(p)              R2 :  0.9752
 (0.04535 +/- 0.00101) * p                        R2 :  0.5675
 (0.5881 +/- 0.4871)                              R2 :  0.0013
```

Figure 5.2: Example Output of `lca`

selecting a model for Load Imbalance, while varying p. The tool has 3 default models that describe how Load Imbalance often varies with p: $p\sqrt{p}$, p, and null (independent of p). The R^2 column shows that $p\sqrt{p}$ is the best-fitting model, and the form column indicates that the coefficient based on a least squares fit of these data is 0.01022.

The R^2 value provided by `lca` is the fraction of the total variation in the measurements that is explained by each model. Generally, the user can select a model if it explains a large fraction of the total variation (*e.g.*, more than 95%). Each parameter estimated by `lca` is also given a 90% confidence interval. The confidence interval is provided so that the user can distinguish terms in each model that do not contribute to goodness-of-fit; if any parameter's confidence interval contains zero, then that parameter cannot be statistically differentiated from zero, and the associated term should be eliminated. Eliminating terms from a model is useful because it narrows the confidence intervals on the model's predictions.

`lca` can also provide graphical help in evaluating the goodness-of-fit of each typical behavior. By supplying the `-g` option, `lca` creates input for the plotting program GNUPLOT. Graphical output of `lca` for the example used in Figure 5.2 is shown in Figure 5.3.

All of the typical behaviors used by `lca` are read in from text files, allowing easy update and modification by the user as new understanding of typical behaviors is developed. In addition, to aid in exploring various candidate functional forms, any formulae supplied by the user on the command line will also be tested for

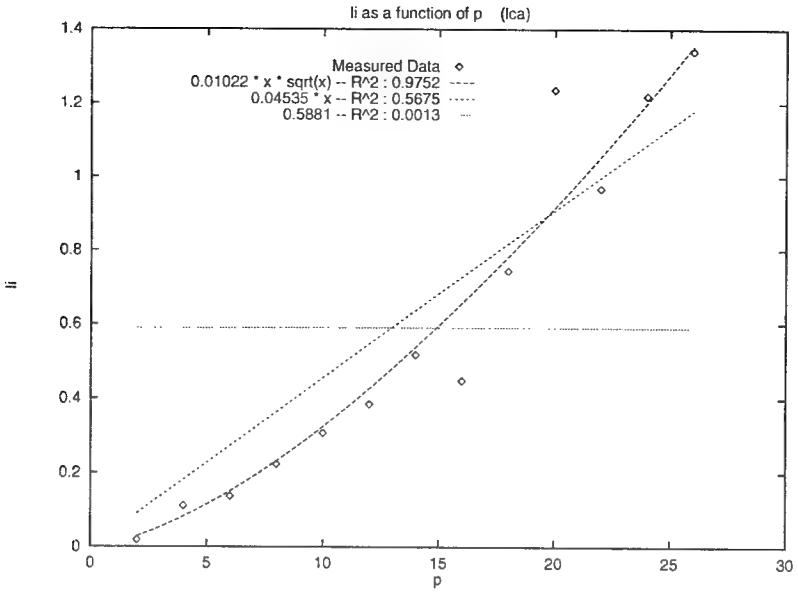


Figure 5.3: Example Graphical Output from `lca`

goodness-of-fit and compared to the default forms by `lca`.

Internally `lca` uses a direct method of general linear least squares minimization based on Singular Value Decomposition (SVD) [Press *et al.*, 1988]. Direct methods such as SVD are limited to fitting curves which are a linear combination of functions of x. Thus direct methods can find the values for the constants (k's) in expressions such as $k_1 x \log(x) + k_2 x^2 + k_3 \epsilon^x$, but cannot typically do so for expressions such as $k_1 e^{k_2 x}$. To solve nonlinear expressions it is usually necessary to use iterative methods, but fortunately the expressions used in modeling parallel overheads are normally quite simple and are characteristically linear in x.

5.2 Lost Cycles Analysis

The previous section showed that overhead decomposition significantly assists in the process of performance modeling because, once decomposed, overheads can be seen to have simple behaviors. This section shows how overhead decomposition can be exploited in practice.

Lost cycles analysis of an application involves three main steps, shown in Figure 5.4 for a hypothetical program named `runfast`. The remainder of this section describes in detail the steps shown in the figure.

Program measurement for lost cycles analysis is performed using the predicate profiler `pp`, as shown in Step 1 of Figure 5.4. Output from the predicate profiler is

1. For each environment variable $V \in E$ **[Measure]**
   (a) For a small number of values of V
      i. `run runfast`
      ii. `pp -l "`V`=<value>" >> runfast.pplog`
2. For each category $C \in (IP, LI, SL, CL, RC, PC)$ **[Model]**
   (a) For each environment variable $V \in E$
      i. `lca -v` V `-c` C `-f runfast.pplog -x <extr-expr>`
      ii. Select a resulting functional form F_V
   (b) Form all functions $F_1, F_2, ...$ whose projections are the F_V's
   (c) `lca -v` E `-c` C `-f runfast.pplog "`$F_1, F_2, ...$`"`
   (d) Select a resulting model for $C(E)$
3. Form the program's performance function as $T_P(E) = \sum_C \frac{C(E)}{p}$. **[Sum]**

Figure 5.4: Overview of Lost Cycles Analysis Steps

stored in a single datafile that eventually contains all of the needed performance data for the application.

Useful measurement of the application under study requires proper experimental design [Atkinson and Donev, 1992]. In order to determine when environmental factors *interact*, that is, are not strictly additive, the user must use a factorial or reduced-factorial design [Box *et al.*, 1978]. These designs are suitable for assessing interaction because they do not vary factors in isolation; instead they vary each factor over a range of settings of the other factors. In practice a factorial design can be accomplished if the user samples the "edges" of the parameter space, for 3 or more points on each edge. For a large number of environmental factors, however, this can result in the need for many measurements. In such high-dimensional cases, the number of measurements can be decreased by using a reduced-factorial design, at a small cost in accuracy and complexity of experimental design.

Step 2 of Figure 5.4 shows that the model construction for each overhead category is performed separately. In this step we refer to the T_c component of the application's execution as "Pure Computation" (PC) and treat it as another performance measurement to be modeled.

Step 2(a) selects a projection for the overhead category along each axis of E. This is the principal use of the `lca` tool in our approach. For the overhead category C and the external factor V, `lca` performs a least-squares fit of all typical behaviors to the measured data, as described in Section 5.1.2. Based on the R^2 values output from `lca`, and using the graphical presentation of each fit if desired. the user selects a functional form to describe the behavior of category C as a function of V.

For some combinations of categories and variables, there can be no typical behaviors. For example, the relationship between Pure Computation and problem size is entirely application-dependent. In these cases simple static analysis provided by the programmer will be sufficient. For $PC(n)$, the programmer need only determine the big-O running time expected from the serial algorithm. For example, for LU Decomposition, a simple analysis of the triply-nested loop would lead the programmer to expect running time to be modeled by a third-order polynomial in n (a side of the matrix). The algorithmic constants need not be determined; instead the programmer uses a third-order polynomial as input to `lca`, which then returns the necessary constants, incorporating in the process both algorithmic constants and machine parameters.

After Step 2(a) the user knows what the projection of the performance function for category C is for each external factor in E — the *univariate* models. In Step 2(b) the user then must develop a single performance function that has as its domain all the external factors in E – the *multivariate* model. In many cases, external factors interact. Assessing which factors are additive and which factors are multiplicative (interacting) is done using the methods of multivariate data

analysis [Jain, 1991, Part IV]. The basic method is to form terms which represent all posible interactions of environmental factors, and then assess the amount of variation in the data that is explainable by each of the terms. Terms that explain large amounts of variation in the data represent significant interactions, and terms that do not explain much variation can be ignored.

This analysis can be done using `lca` by forming functions F_x that represent the various possible combinations (via addition or multiplication) of the univariate models. These functions, developed in Step 2(b), are then compared for their relative goodness-of-fit. Steps 2(c) and 2(d) use `lca` to compare the functions, and the user selects the overall performance function for the category that best fits the measured data.

Once a performance function has been selected for each category, the performance model of the entire application is constructed (Step 3) by simply summing the expressions, and dividing by the number of processors used (p), since all overhead measurements were originally taken as sums over all processors.

Although the process outlined in Figure 5.4 contains a number of iterative steps, the iterations will usually be limited. The primary reason that iterations will be limited is that we are often concerned with only two environmental factors, n and p, so Steps 1 and 2(a) will only loop twice.

5.3 Using Lost Cycles Analysis

This section presents three case studies of the use of lost cycles analysis. First we show how lost cycles analysis can help construct accurate analytic models of program scalability based on a small number of measurements. Then we present two examples that illustrate how lost cycles analysis can help solve the problem of selecting the best parallelization for a program by comparing lost cycles models.

5.3.1 Modeling the Performance of 2D FFT

The ability to capture the expected performance of a program based on a small number of measurements is critical to managing the problem of understanding and selecting among differing implementations. Measuring and debugging program performance without gathering large amounts of data is an important capability in its own right, and is the subject of much current effort [Ball and Larus, 1992; Hollingsworth and Miller, 1993; Miller and Choi, 1988; Netzer and Miller, 1992]. The results in this section show that lost cycles modeling is a convenient way of capturing large amounts of performance data, requiring minimal measurement effort and little storage.

In this example we will:

Table 5.2: Models of Overhead for 2D FFT as Functions of n and p

Category	Abbrev.	Model	
		Varying n	Varying p
Pure Computation	PC	$n^2 \log(n)$	1
Load Imbalance	LI	$n \log(n)$	$p\sqrt{p}$
Insufficient Parallelism	IP	1	p
Synchronization Loss	SL	0	0
Communication Loss	CL	n^2	p
Resource Contention	RC	n^2	$p, p > \theta$

1. Measure the program's lost cycles for two cases: the highest-overhead case, and the lowest-overhead case.

2. Select appropriate simple models for each category of lost cycles and for pure computation, as separate functions of varying data size and varying number of processors.

3. Use the lost cycles measurements to parameterize the models selected, yielding predictions for running time over the entire range of data sizes and numbers of processors.

We selected the highest- and lowest-overhead data points for measurement because by measuring an execution with high relative overhead we get an accurate estimate of true overhead, and by measuring an execution with low relative overhead we get an accurate estimate of pure computation. Rules of scalability analysis guide us in selecting the data points for measurement: in a parallel system, overheads tend to grow with increasing processors and decrease with increasing data size. These observations suggest that we should capture lost cycles for an execution with maximum processors and minimum data (highest overhead) and for an execution with minimum processors and maximum data (lowest overhead).

The simple models we chose to describe each overhead are listed in Table 5.2. as separate functions of n (which in this section represents the length of a side of the input matrix) and p (number of processors). Each model has an implicitly associated constant; the purpose of our lost cycles measurements is to discover the constants. Each of these models is a simple, initial approximation to reality. Better models for each are possible, but not necessary in this context since they trade increasing accuracy for increasing measurement cost, and decreasing analytic tractability.

Considering first the models for varying n, the model for pure computation is based on simple algorithmic analysis of 2D FFT. The model for load imbalance is

based on the length of an iteration of the program's parallel loops. There are no synchronization operations in the program, so we expect no synchronization loss. The model for insufficient parallelism is based on the portion of the code that runs serially, which has no data size dependencies. The model for communication loss is based on the total amount of data used. Finally, the model for resource contention is based on the expectation that resource contention will be proportional to data size.

In choosing models of overhead as we vary the number of processors, we can rely on the large body of work reviewed in Section 2.2.2 to provide likely candidate models. Most of the models we use are straightforward: pure computation does not vary as we vary processors, insufficient parallelism obeys Amdahl's Law [Amdahl, 1967], and synchronization loss is zero.

Load imbalance can arise in two ways: variation in the running time of each loop iteration, and unequal numbers of loop iterations handled by different processors. If variation in running time of iterations is random, the time taken by the longest iteration can be modeled using order statistics (*e.g.*, [Hummel *et al.*, 1992]) and predicted to grow proportionally to $\sqrt{p}$. Communication loss for this application is proportional to p. Finally, resource contention can be expected to grow linearly once the number of processors passes a threshold value.

Using the lost cycles measurements from the two executions we then parameterize the six models (*PC, LI, IP, SL, CL,* and *RC*). For example, the final form for Load Imbalance is:

$$LI(n,p) = \frac{n \log(n) p \sqrt{p}}{36200}.$$

Using the basic identity $T_P(n,p) = (T_c(n) + T_o(n,p))/p$, we construct the performance model for the implementation as:

$$T_P(n,p) = \frac{PC(n,p) + LI(n,p) + IP(n,p) + SL(n,p) + CL(n,p) + RC(n,p)}{p}$$

The results for the 2D FFT program are shown in Figure 5.5. These plots show the performance of the application measured in Mflops, as a function of both number of processors and of dataset size. The upper plot shows the predictions of our model for 78 data points, that is, all points within the range of processors and data set sizes we set out to model. The lower plot shows the actual measured performance of the application on the KSR1 for those same 78 data points.

As can be seen, the model is an idealized but reasonably accurate approximation to actual performance. In fact, the average relative error of the model with respect to the actual performance, over all 78 points, is only 12.5%. For comparison, the average relative error of a simple linear interpolation based on a least squares fit of the four "corner" points is over 750%. Thus both the overall shape of the predicted performance curve and its actual values are sufficiently accurate

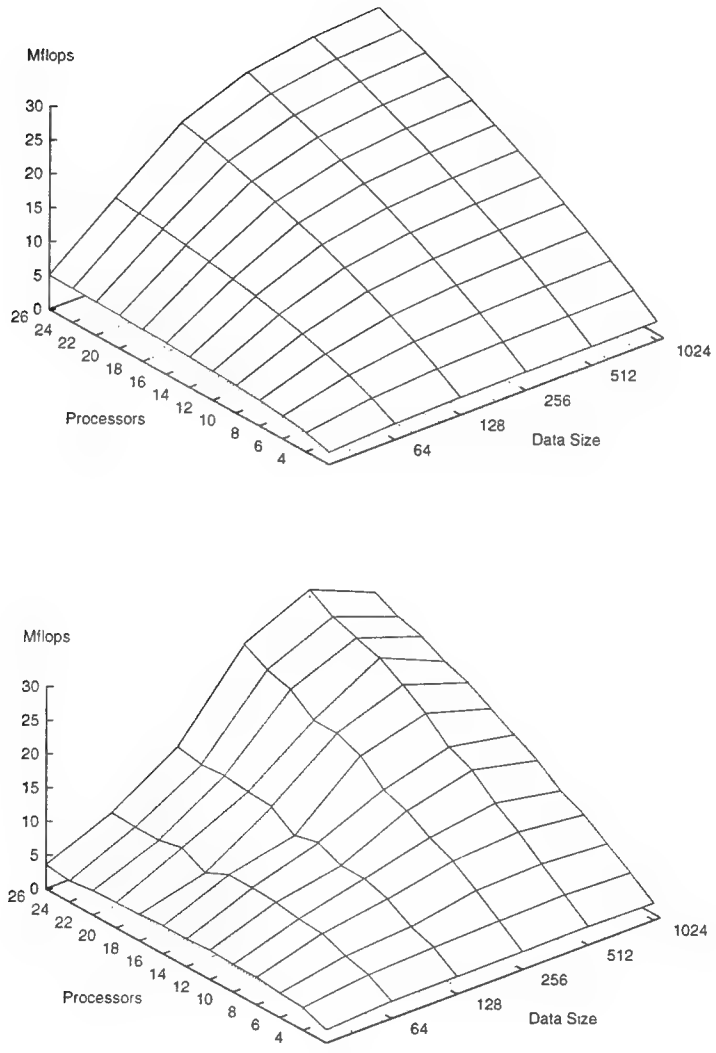


Figure 5.5: Predicted (top) and Actual (bottom) Performance of 2D FFT

to allow it to be used in studying tradeoffs against an alternative implementation, which we will do in the next section.

5.3.2 Task Parallel vs. Data Parallel 2D FFT

A comparison of the task parallel and data parallel implementations of 2D FFT on the iWarp was presented in [Subhlok *et al.*, 1993]. On that machine, the authors discovered that as data set sizes are varied past a certain threshold, the choice of which implementation is best changes. For small data sets ($n \leq 128$) the parallel tasking implementation outperformed the pure data parallel implementation. For large data set sizes ($n \geq 256$), the purely data parallel implementation outperformed the parallel tasking implementation. The principal reason for this effect is that in the parallel task version, communication between tasks must pass through a single channel of the iWarp network, while purely data parallel communication can take place along multiple channels. For small data sizes, the larger problem granularity of parallel tasking leads to better performance, but as the problem size increases, intertask communication becomes a bottleneck.

It is interesting to ask whether a similar effect would be observed when this application is run on the KSR1, a machine with a significantly different architecture. Unfortunately, the data from the iWarp cannot help us decide which executions to measure, since the machines are so different. Thus we immediately run into a problem: perhaps there is a crossover between implementations in some section of the environment space (here, n and p), but finding the crossover would require measurements over the entire space.

To answer this question using the lost cycles approach, we only need to construct a model for the application. The simplest approach in this case is to 1) decide whether the category models used for the pure data parallel implementation follow the same functions as did the task parallel models; and 2) determine new constants for the overhead functions. To do this we measured 6 points varying the data set size (to explore the functions of n) and 6 points varying the number of processors (to explore the functions of p).

The results are shown in Table 5.3. The table shows the functions and the associated constants for the variable n, since the models did not differ significantly in the p dimension.[1] These functions immediately answer our questions about these two implementations. First of all, resource contention in the task parallel implementation is significantly less than in the data parallel implementation, indicating that the channel bottleneck effects observed on the iWarp will not be present on the KSR1. This conclusion is reasonable, since intra-ring communication costs are insensitive to source and destination on the KSR1. In fact, we see that resource contention is only about half as great in the task parallel

[1] We hold p constant at its maximum value (26) in these formulae.

Table 5.3: Performance Models for Data Parallel and Task Parallel 2D FFT

Category	Data Parallel	Task Parallel
Pure Computation	$\frac{n^2 \log(n)}{3550}$	$\frac{n^2 \log(n)}{3350}$
Load Imbalance	$\frac{n \log(n)}{63.0}$	$\frac{n \log(n)}{81.9}$
Insufficient Parallelism	3.36	$\frac{n^2}{992}$
Synchronization Loss	0	$\frac{n^2}{31600}$
Communication Loss	$\frac{n^2}{14900}$	$\frac{n^2}{12200}$
Resource Contention	$\frac{n^2}{20100}$	$\frac{n^2}{35600}$

version, since in the pure data parallel version, all processors are simultaneously requesting *and* providing data during the matrix transpose, while in the parallel task version, half the processors request data and the other half provide it.

The second observation is that on the KSR, the task parallel implementation will always perform more poorly than the pure data parallel implementation. Synchronization loss and insufficient parallelism are functions of n^2 in the task parallel implementation. The reason for this change from constant values to functions of n^2 when the implementation is changed can be seen in observing that synchronization loss is now equal to about a third of the communication loss. In fact, in this implementation, the two tasks do *not* incur equal overhead. The task that transposes the matrix incurs more overhead because it must traverse the source matrix across cache lines, destroying locality. Thus each loop iteration for the transposing task takes slightly longer than an iteration of the initializing task; a pair of spin locks prevents either task from overtaking the other. As a result of this synchronization loss in the main thread of the initializing task, the other threads in its group must wait without work, incurring lost cycles due to insufficient parallelism. This insufficient parallelism has a particularly small constant in the denominator and hence dominates the small improvements in resource contention and load imbalance generated by task parallelism.[2]

Thus we have quickly answered the question of whether two implementations, known to have a performance tradeoff on at least one architecture, have a similar

[2]Presumably these effects were not present on the iWarp because of message passing optimizations.

Table 5.4: Seconds of Pure Computation and Wasted Speculation in Subgraph Isomorphism

Category	Processors						
	1	2	3	4	5	6	7
Wasted Speculation	0.00	50.1	103	10.7	14.2	1.25	1.44
Pure Computation	51.4	50.1	51.7	3.56	3.52	0.227	0.214

performance tradeoff on the architecture of interest to us. To do this, we only needed to measure a small number of data points in each of the two environmental dimensions, and compare the resulting lost cycles models.

5.3.3 Subgraph Isomorphism

We now return to the example of subgraph isomorphism. Referring to Figure 3.1, we would like to understand the differences among the four parallelizations of subgraph isomorphism (tree, tree plus instruction, loop, and loop plus instruction) as a function of p. Since we are only studying one dimension of the environment in this case, we can measure the implementations over the entire range and use the resulting models to explain their relative performance.

In order to achieve completeness for this application, we need to include measurement of cycles lost in wasted computation (due to fruitless speculation). We will define wasted computation in this case as the processor cycles spent searching a subtree under the root in which no solutions are found. Modeling this category is difficult because of the effects of speculative parallelism, but since we are only searching a single dimension, we can simply measure lost cycles due to wasted computation for the points of interest.

First of all, we show the wasted speculation and pure computation data for the tree parallel case in Table 5.4. This table immediately explains why the tree parallel versions outperform the loop parallel versions when $p > 3$, yet do worse than loop parallel versions when $1 < p < 4$. The pure computation required to solve the problem changes drastically with increasing p because these executions of the program are searching for only one solution — the first processor to find a solution ends the computation. Clearly, the subtrees searched by processors 2 and 3 do not yield the solution, since wasted computation increases in steps of 50 seconds (the time spent by processor 1 in finding a solution). Processor 4 finds a solution in its subtree much sooner than the others however, and the effect is repeated again by processor 6. The data in this table shows that even

Table 5.5: Lost Cycles Models for Two Implementations of Subgraph Isomorphism

Category	Implementation	
	Loop	Loop + Instr.
Pure Computation	54.7	55.9
Load Imbalance	$\frac{p\sqrt{p}}{.750}$	$\frac{p\sqrt{p}}{1.46}$
Insufficient Parallelism	$\frac{p}{.496}$	$\frac{p}{2.08}$
Synchronization Loss	0	0
Communication Loss	$\frac{p}{.110}$	$\frac{p}{.233}$

when overhead categories are difficult to model, the raw lost cycles data can be informative in ways that are difficult for tools that do not provide completeness.

Next, we consider why loop and instruction parallelism together outperform loop parallelism alone. Since there is no speculative computation in the loop parallel executions, this is done most easily by considering the lost cycles models, which are shown in Table 5.5.[3]

We might expect that the benefit of adding instruction parallelism to the implementation would be in decreased pure computation (since instruction parallelism shows up as decreased pure computation). However, the lost cycles models show that in fact, pure computation is relatively unchanged between the two implementations. The model shows that the additional cost of packing and unpacking data counterbalances the pure computational decrease gained in parallel set operations, actual increasing pure computation slightly.

In fact, the gains from adding instruction parallelism come from a less expected direction. First of all, by packing datasets, the overall data being transferred decreases, decreasing communication loss by a factor of 2. Secondly, load imbalance within loops and insufficient parallelism are decreased, since these overheads both tend to increase as communication increases.

The lost cycles models in Table 5.5 also explain why the loop parallel implementation benefits from the addition of instruction parallelism, while the tree parallel implementation actually suffers slightly from the same addition. Since the tree parallel version searches separate subtrees in parallel, it has essentially no communication loss (this effect can be observed from the lost cycles data as well). The absence of communication loss in the tree parallel implementation

[3]These data are derived from the SGI implementation of the predicate profiler, which did not support the measurement of resource contention.

means that it cannot reap the benefits of instruction parallelism, whether directly in decreased communication, or indirectly through decreased load imbalance and insufficient parallelism. Instead the tree parallel version only pays the (small) price of instruction parallelism, a fact reflected in the performance data shown in Figure 3.1.

In this example we have gained a number of insights into the relationship between the particular implementation of the subgraph isomorphism program and its corresponding performance. These insights were gained from measurements of only 12 data points, showing the power of lost cycles analysis to provide tuning guidance — especially since these insights were not evident from our original collection of over 37,000 data points.

5.4 Summary

In this chapter we have shown a method for constructing an application's performance model based principally on dynamic measurements. For each external variable, we empirically determine its effect on parallel overhead. This determination is accelerated by a curve fitting tool incorporating *a priori* knowledge of typical overhead behaviors. The application's performance model can then be composed from the effects of the component overheads.

Most previous performance prediction methods incorporate a static analysis component to help model external factors. Our approach can be seen as an attempt to extend dynamic analysis as far as possible in solving this problem. The benefit of using dynamic analysis arises from the use of the machine itself to identify significant performance effects. For example, using static performance prediction to accurately predict the costs of cache conflicts, register management, and communication traffic can be quite difficult and expensive. Dynamic analysis allows us to capture these costs accurately without significant effort or expensive simulation.

Lost cycles analysis serves as a bridge between mathematical modeling, such as scalability analysis, and performance measurement. It extends the power of the user to generalize from performance measurements, and it provides a measurement tool for the analyst to ground analyses in experiments. However, the dynamic approach taken by lost cycles analysis cannot assist the user directly in predicting the effect of program modifications. The next chapter describes our method of static analysis, which builds on lost cycles analysis to provide the user with a complete performance prediction capability.

6 Predicting The Effects of Varying Program Structure

The previous chapter showed how to construct a performance model $T_P(E)$ principally from dynamic execution data, using the technique of lost cycles modeling. However, as described in Section 1.2, the goal of performance prediction is to build the function $T_P(I, E)$, where I denotes the internal performance factors of the application. That is, to be truly useful to programmers, a performance prediction method must estimate the performance of a range of potential implementations. In this chapter we show a method for constructing performance models that are functions of both implementation and environment; we call our method *lost cycles modeling.* Lost cycles modeling has two parts: first lost cycles analysis is used to form models for portions of the application, which we call code *fragments.* Next, the lost cycles models for the program's fragments are composed into an overall application model using static analysis of the program's source code.

As discussed in Chapter 3, internal factors include the type of parallelism used (*e.g.*, task parallelism, data parallelism, or vector parallelism) and the choice of which code to parallelize (*e.g.*, how many of the program's loops to parallelize). Lost cycles modeling takes a static approach to building performance models based on program structure. Lost cycles modeling can construct performance models based on program structure because of the utility of lost cycles analysis for code fragments. Since lost cycles analysis describes the performance of code fragments in terms of external factors, it captures the information necessary to describe how components of an application will behave when reorganized into a new parallel structure. This greatly simplifies the static analysis required to construct the new implementation's performance model.

6.1 Constructing Performance Models

Lost cycles modeling avoids complex static analysis by using the tools described in the last chapter to empirically discover the performance of code fragments. In the next two sections we describe how lost cycles modeling constructs

performance models; first we describe the notion of code fragments more formally, then we describe how lost cycles modeling composes fragment models into application models.

6.1.1 Code Fragments and Parallel Structure

Lost cycles modeling operates on applications with primary emphasis on their parallel structure. We will define parallel structure as a particular hierarchical composition of the application's source code. The hierarchy can be represented as a directed acyclic graph (DAG); internal nodes of the graph represent either sequential (depth-first, left-to-right) or parallel execution of their children. Leaves of the graph each represent some contiguous portion of the application's source code; the graph is similar to the static basic-block representation of a program typically used by compilers [Aho *et al.*, 1988]. We require that the children of a node be homogeneous — that is, all internal nodes or all leaf nodes. This framework, while quite simple, is adequate for the static analysis performed by `lcm` because of the performance annotations we will associate with certain internal nodes of the graph.

We will refer to nodes whose children are leaves as *penultimate* nodes. We make the homogeneity requirement so that all the children of any penultimate node are leaf nodes, which ensures that we can clearly identify the points in the tree at which to perform lost cycles analysis. The source code represented by each penultimate node is a code *fragment*. Given this framework, we can state the lost cycles modeling process more precisely. Lost cycles modeling consists of performing lost cycles analysis for each penultimate node, followed by composition of the resulting fragment models into a program model. The benefit of lost cycles modeling derives from the flexibility of program rearrangement possible once the various fragment models have been built: any alternative parallel structure using those fragments can be evaluated automatically by `lcm`.

As an example, consider the (edited) source code for task parallel 2D FFT shown in Figure 6.1 and Figure 6.2. Figure 6.1 shows the main routine of the program and Figure 6.2 shows the important subroutines. In the figures we have edited out the synchronization operations and the body of the computation for clarity. Comments preceded by `c*ksr*` are compiler directives that introduce parallelism [Kendall Square Research, 1992]. The directive `c*ksr* parallel sections` creates parallel tasks, each of which is introduced by the directive `c*ksr* section`. The directive `c*ksr* tile` creates a parallel loop, with a variety of possible scheduling strategies; in this example the strategy is `slice`, which statically schedules the N loop iterations in chunks of size N/p.

A parallel structure for task parallel 2D FFT is shown in Figure 6.3. In the figure, node 1 represents the entire program — it is a *serial* node. Likewise, serial

```
c-----------------------------------------------------------------
      program main
      real a(n,n,2,iters)
      call initialize_program
c*ksr* parallel sections
c*ksr* section
      do k=1,iters
         call initialize(a(k))
         call cffts(a(k))
      end do
c*ksr* section
      do k=1,iters/2
         call transpose(a(k))
         call cffts(a(k))
      end do
c*ksr* end parallel sections
      call print_performance
      end
c-----------------------------------------------------------------
```

Figure 6.1: Edited Source for the Main Routine of Task Parallel 2D FFT

nodes 2 and 4 represent the execution of their (single) children. Node 3 represents the task parallelism created by the `c*ksr* parallel sections` directive — it is a *parallel* node. Nodes 5 and 6 are serial nodes corresponding to the two tasks, each of which contains two serial subroutine calls. Nodes 7 through 10 are parallel nodes, each corresponding to a parallel loop created by the `c*ksr* tile` directive. Nodes 8 and 10 are in fact the same node, making the true graph a DAG rather than a tree; they are only shown as separate nodes in the figure for clarity.

To perform lost cycles modeling on Task Parallel 2D FFT, the programmer uses lost cycles analysis to construct performance models of the penultimate nodes, which are 2, 4, 7, 9 and 8/10. The programmer then uses the tool `lcm` to automatically compose those models into an overall model of application performance, based on a parallel structure such as the one in Figure 6.3.

As long as the structure of the DAG meets the homogeneity requirement, its details are up to the user, and lost cycles modeling can be used to construct a performance model of the application. For example, the entire 2D FFT application could be represented by a root node and a single leaf; in this case the process would degenerate into the lost cycles analysis described in the previous chapter. Naturally, the user would like to minimize the number of penultimate nodes, since each one must be analyzed separately using lost cycles analysis. However, to be maximally useful in exploring alternative parallelizations, the DAG should

```
c-----------------------------------------------------------------
      subroutine transpose(a)
c*ksr*tile(k,strategy=slice)
         do k=1,n
      [ ... parallel loops performing matrix transposition ... ]
         enddo
c*ksr*endtile
      return
      end
c-----------------------------------------------------------------
      subroutine initialize(a)
c*ksr*tile(j,strategy=slice)
         do j=1,n
      [ ... parallel loop performing matrix initialization ... ]
         enddo
c*ksr*endtile
      return
      end
c-----------------------------------------------------------------
      subroutine cffts(a)
c*ksr* tile(col,strategy=slice)
      do col=1,n
      [ ... parallel loop -- each iteration performs a 1D fft ... ]
         enddo
c*ksr*endtile
      return
      end
c-----------------------------------------------------------------
```

Figure 6.2: Edited Source for Subroutines of Task Parallel 2D FFT

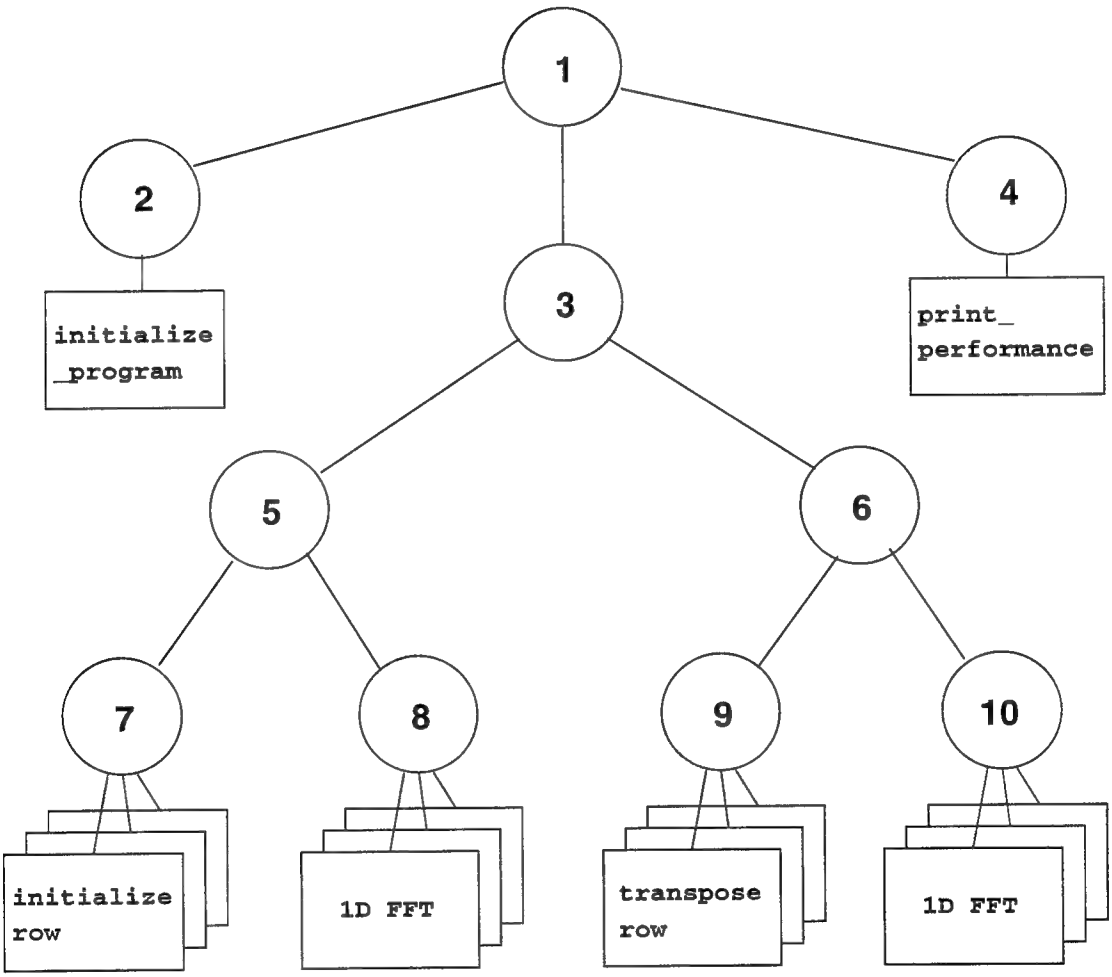


Figure 6.3: Parallel Structure for Task Parallel 2D FFT

contain penultimate nodes whose leaves are not likely to be candidates for parallel decomposition. These two goals are in tension, but can be satisfied as follows. In practice, identifying the penultimate nodes in a program only requires the programmer to 1) identify all the sources of parallelism the user expects to explore; and 2) form a partition of the serial code in the source that does not mix source lines from different sources of parallelism in the same fragment.

This principle is illustrated in Figure 6.3. Nodes 5 and 6 could have been chosen as penultimate nodes. Then lost cycles analysis need only have been performed on the two tasks within the parallel section, rather than on the three loops. However, by decomposing nodes 5 and 6, the user has the ability to consider a wider range of alternative parallel structures — including one which runs each loop in its own task, so that all four loops execute at once.

6.1.2 Composing Fragment Models

In order to compose models of code fragments into a model for the entire application, two steps are needed. First, the relationship between external variables of the application and those of the fragment models must be determined. Then the additional or decreased overheads incurred due to the relationship between fragments at runtime must be estimated.

The first step consists of determining how the values of external variables vary for the component models. This is demonstrated graphically by comparing Figure 3.2 with Figure 3.3. Both figures represent the execution of 2D FFT on six processors. However, in Figure 3.2, the number of processors (value of p) used in the component models is six, while the value of p for the fragment models in Figure 3.3 is three. The difference occurs because the task parallel 2D FFT employs two parallel loops simultaneously; these loops must share the available processors. In comparison, the values of n for the fragment models in both parallel structures are equal.

The second step consists of determining any additions or subtractions necessary to the parallel overheads predicted by the modified fragment models. For example, task parallel 2D FFT can incur load imbalance and additional communication between the two tasks that does not occur in data parallel 2D FFT. In addition, the need for the transpose task to wait for the first array to become available before it can begin work (a type of pipeline filling delay) creates synchronization loss that does not exist in data parallel 2D FFT. The approach used by `lcm` to implement these two steps will be described in Section 6.2.2.

The overheads added or subtracted during composition are expected to typically be small in relation to the overheads accounted for in the component lost cycles models. For example, the lost cycles model for a program fragment consisting of a parallel loop will contain predictions for the communication occcuring

between loop iterations. Only the communication required to initially access the data items used will be added during the composition step.

In summary, given the ability to build fragment models and then to compose fragment models into application models, lost cycles modeling can provide the programmer with considerable power to experiment with alternative parallel structures for an application, without fully implementing all the alternatives. Lost cycles modeling can achieve this result in part by not attempting to provide performance prediction capabilities for arbitrary restructurings of source code; instead it relies on the programmer to identify the range of reasonable program structures. However, once a set of code fragments have been identified, any restructuring of those fragments is suitable for analysis using lost cycles modeling. In the next section we describe tools to automate this process.

6.2 Lost Cycles Modeling

Lost cycles modeling is supported by modifications to the tools `pp` and `lca`, and by a tool currently under development, `lcm`. In the next section we describe the modifications we have made to our existing tools to support lost cycles modeling, and then we present design details for the tool `lcm`.

6.2.1 Tool Support for Source Code Annotations

We made two modifications to our existing tool set to provide support for lost cycles modeling. We modified `pp` to profile code fragments, and we modified `lca` to output models in a format for use in annotating source code.

Our modifications to `pp` allow the user to specify each code fragment in the source, and identify it with an integer tag. Identifying a code fragment is accomplished by simply bracketing it with calls to library functions, and providing the value of the tag. `pp` was also modified to accept the tag value of interest as a command line parameter. Thus one execution of the application can be used for multiple runs of `pp`, each profiling a different portion of the execution by specifying a different tag. The presence of both code-bracketing tags and synchronization events in the event log allows `lcm` to perform critical path analysis when necessary, as discussed in the following section.

Since the tag is a numeric value, it is included in each record of the application's profile data, allowing the profiles for all code fragments to be placed in the same file. `lca` can be used to construct models for each profile by specifying the value of the tag in the extraction-expression via the `-x` option.

Finally, `lca` can be used to output a performance model in the form of source code comments using Fortran 77 syntax. These models can be cut-and-pasted di-

rectly into the application's source; as the code is rearranged to explore alternative representations, the performance models are moved with the code fragments.

6.2.2 Tool Support for Lost Cycles Modeling: `lcm`

The primary role of `lcm` is in composing fragment models into an application model. The approach taken by `lcm` to compose models is presented in this section.

As described in section 6.1.2, `lcm` has two steps to perform. In the first step, `lcm` must determine the relationship between external variables of the application (*e.g.*, n and p) and those of the fragment models. In the second step, `lcm` must estimate the increased or decreased overheads incurred due to the relationship between fragments at runtime.

`lcm` handles the first step using default rules, which can be overridden by the user. Rules are applied from the root node downward in the DAG. For example, the default rule for the creation of T parallel tasks is that each parallel task has p/T processors assigned to it. To modify the defaults, the user simply places an annotation on the code fragment specifying the value of p as a constant, or symbolically (*e.g.*, $p/T - 1$).

The default rules for external variables provide new values for the inputs to each code fragment's lost cycles model. The ability to use simple rules in this step provides an example of the value of separating the static and dynamic components of performance prediction along the boundary between internal and external factors. Static analysis can easily determine the new values of the external factors, while the results of dynamic analysis to determine the effects of external factors can simply be reused.

To handle the second step, `lcm` uses more complex processing. Models are composed from the leaf nodes upward in the DAG. The method of modifying overhead predictions is specific to the category of overhead, and the type of composition occurring at the node (serial or parallel). Three methods are used: structural rules, data traces, and synchronization traces.

Structural Rules

Structural rules handle the Load Imbalance and Insufficient Parallelism categories of overhead. The structural rules for serial composition don't introduce any modifications to overhead. For parallel composition of T tasks, the structural rule states that load imbalance (LI^+) is added according to the formula:

$$LI^+(n,p) = \sum_{t \in T} \max_{t' \in T}(T_{P,t'}(n_{t'}, p_{t'})) - T_{P,t}(n_t, p_t)$$

where $T_{P,t}$ is the lost cycles fragment model for task t, and n_t and p_t are that model's modified inputs. This formula states formally the commonly understood definition of load imbalance as the time spent by all tasks while waiting for the last task to finish. For parallel composition, no modifications to Insufficient Parallelism are necessary.

Data Traces

Data traces handle the Communication Loss and Resource Contention categories of overhead. Currently, input to the structural rules are simple data use annotations supplied by the user. These data use annotations could supplied in large part by compiler analysis, but for proof-of-concept purposes user annotations are easily enough supplied. The annotations specify which variables are read and written by each code fragment, and are similar to those used to assist parallelizing compilers [Rinard *et al.*, 1992; Subhlok *et al.*, 1993].

Examples of the data trace annotations are shown in Figure 6.4. `lcm` tracks the movement of data between code fragments based on the data dependencies resulting from the user-supplied data annotations. First `lcm` converts the DAG into a tree by replicating any subtrees that have more than one parent. Then `lcm` performs a depth-first, left-to-right traversal of the tree's penultimate nodes to track the data movements that would occur between code fragments during execution. Each penultimate node is considered a logically separate location for data. Movement of data from one processor or set of processors to another occurs when two penultimate nodes have a least upper bound that is a parallel node, and a data dependence occurs between the nodes (either a true dependence, an anti dependence, or a write dependence).

For example, when processing task parallel 2D FFT using the structure shown in Figure 6.3 and the annotations shown in Figure 6.4, `lcm` proceeds as follows. The first node visited is 2, which has no data annotations, followed by node 7, which outputs `a(k)`. Next `lcm` vists node 8, whose least upper bound with node 7 is node 5, a serial node, so no communication occurs; however, node 8 acquires `a(k)` since node 8 outputs `a(k)`. Next `lcm` visits node 9, whose least upper bound with node 8 is node 3, a parallel node. In addition, there is a true dependence between node 8 and node 9, so communication of data item `a(k)` occurs at this point. Finally, `lcm` visits node 10, at which no communication occurs, and node 4 which has no data trace annotations.

After determining how data moves among the code fragments in the application, `lcm` estimates the inter-fragment communication in the original source code. Communication Loss and Resource Contention are then added or subtracted from the predictions of the fragment models based on the communication occurring between fragments in the measured executions, and the communication in the application structure under study. The amount of communication loss is predicted

```
c-----------------------------------------------------------------
      program main
      real a(n,n,2,iters)
      call initialize_program
c*ksr* parallel sections
c*ksr* section
      do k=1,iters
c*lcm*   output: a(k)
         call initialize(a(k))
c*lcm*   input: a(k)
c*lcm*   output: a(k)
         call cffts(a(k))
      end do
c*ksr* section
      do k=1,iters/2
c*lcm*   input: a(k)
c*lcm*   output: a(k)
         call transpose(a(k))
c*lcm*   output: a(k)
         call cffts(a(k))
      end do
c*ksr* end parallel sections
      call print_performance
      end
c-----------------------------------------------------------------
```

Figure 6.4: Main Routine of Task Parallel 2D FFT With Data Trace Annotations

as the cost of a single transfer of each data item read. Resource Contention is handled in a similar way, except that the resource contention cost is a function of the number of processors reading the data and the amount of data transferred. The machine parameters needed to predict these costs are measured once per machine.

Synchronization Traces

Synchronization traces handle the Synchronization Loss category of overhead. Essentially, these traces are used to identify when synchronization constraints occur between tasks; if they do occur, `lcm` does not output a closed-form solution to the performance model. Instead, it provides a *critical path analysis* of the application under study.

Critical path analysis [Lockyer, 1964; Yang and Miller, 1988] takes as input a synchronization trace of an application, along with the measured time between

each synchronization event, and outputs the expected running time of the application. Critical path analysis finds the longest sequence of computations dictated by synchronization dependencies.

We perform critical path analysis in the context of lost cycles modeling only if it is determined to be necessary by `lcm`. Synchronization events are captured and correlated with code fragments during the lost cycles analysis phase. The synchronization events generated by each code fragment are stored in trace files, along with their time of occurrence relative to the execution time of the code fragment. To perform critical path analysis, `lcm` tool takes as input the application source, the synchronization traces, and the values of the external variables; it then performs critical path analysis by rescaling and reordering the event traces according to the predicted running time and order of each code fragment in the new environment. The process is similar to that taken for data traces; the DAG is converted to a tree, and traversed to simulate execution.

We have applied the algorithms used in `lcm` by hand to case studies as a proof of concept. These case studies are presented in the next section.

6.3 Lost Cycles Modeling in Practice

To evaluate the potential effectiveness of lost cycles modeling in practice, we have performed it (without the aid of the `lcm` tool) on the task parallel 2D FFT application. We are interested in the comparison of the TP version with the DP version as an typical example of the kind of application restructuring of interest to parallel programmers.

We start by partitioning the program into code fragments. We used a subset of the code fragments discussed in Section 6.1.1 to simplify the manual process. We studied only the code fragments corresponding to nodes 7, 9, and 8/10 in Figure 6.3 — that is, the three different parallel loops in the application.

Next we instrumented the program according to the partition we used — placing delimiters around each of the parallel loops. We then performed lost cycles analysis on the three code fragments. The resulting models were input into the S data analysis system [Becker *et al.*, 1988]. The use of S allows us to specify symbolically the rules for transforming the external variables of the application to the external variables of the component models, and to programmatically construct structural rules such as the load imbalance rule.

Although this section describes the application of default and compositional rules in detail, in practice the application of these rules is handled automatically by `lcm`. When using `lcm`, the user must only provide the program's annotated source code; the corresponding performance model is then output directly by `lcm`.

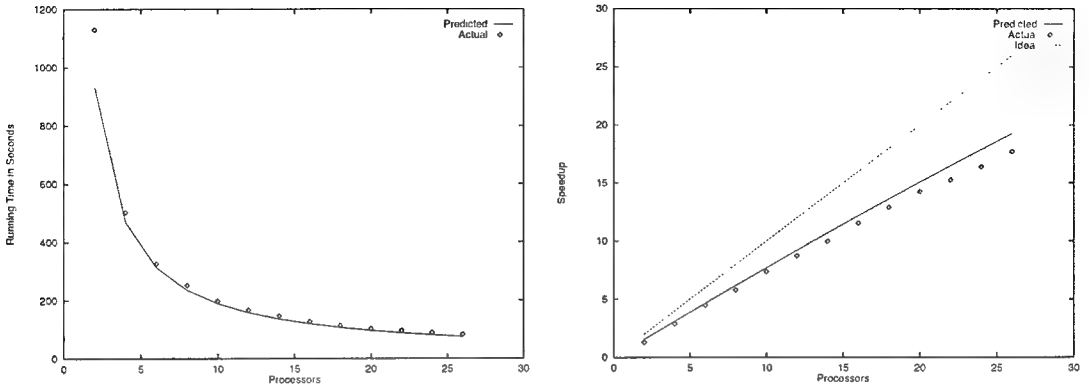
6.3.1 Predicting the Performance of TP 2D FFT

Our first goal was to predict the performance of the TP version itself, using lost cycles modeling. We applied the default rules for p and n, resulting in each loop model taking as its external variables $p/2$ and n. Next we composed the models, working upward from the leaves of the DAG in Figure 6.3.

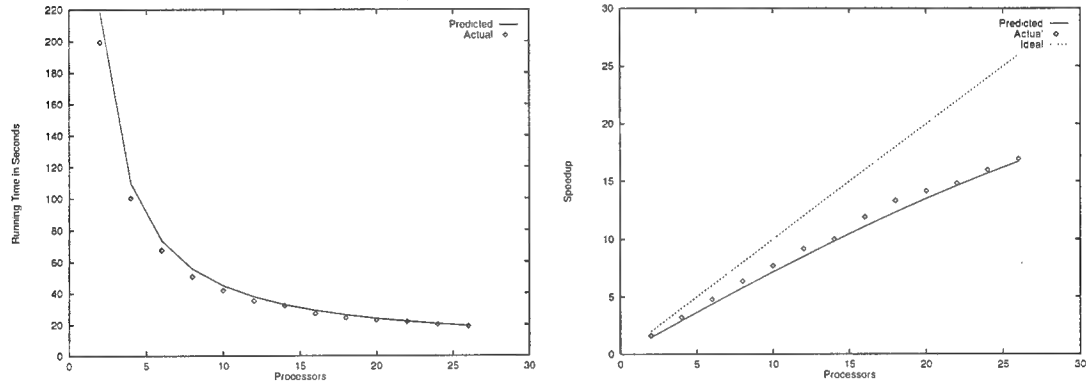
We applied composition rules at internal nodes as follows. Nodes 5 and 6 are serial nodes; they generate no additional overhead, and their composite models are the sums of their component, loop fragment models. Node 3 is a parallel node, and all three composition issues come into play here. Considering structural rule issues, we must apply the Load Imbalance formula, since the predicted running times of the two composite models (for nodes 5 and 6) are different. Second, we must consider data trace issues: node 6 reads in the output of node 5. However, since these communication events in fact did occur in the measured program, they were captured in the fragments' lost cycles models, so no overhead modification is needed for data trace reasons. Finally, synchronization traces show that node 6 must wait for the first matrix to be finished by node 5; this pipeline-fill cost was estimated as the time to execute a single iteration of task 5's main loop. For this reason, synchronization cost was added as the predicted execution time of task 5 divided by the number of iterations it executed — equal to what would be predicted by critical path analysis.

The predictions of the resulting model of task parallel 2D FFT are shown in Figure 6.5, for the three largest data sizes studied. The graphs on the left compare the measured execution time to the predicted execution time, and the graphs on the right compare the measured speedup to the predicted speedup. Note that although we compare model predictions to measured execution time, these measurements were not an input to the lost cycles modeling process; only the three component lost cycles models were used.

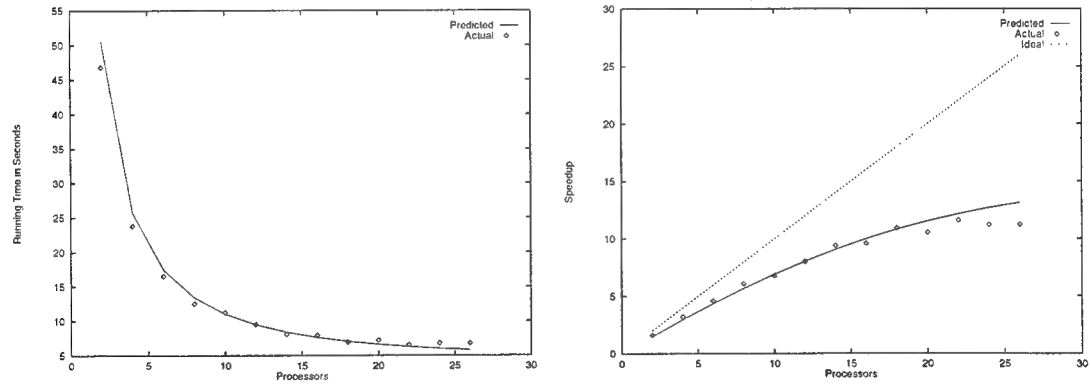
Figure 6.5 shows that the lost cycles model of the application is quite accurate for large values of n. Accuracy on large values of n is important since these values represent the most computationally challenging problems. However model predictions are not quite as accurate for small n. Figure 6.6 shows lost cycles model predictions for $n = 64$. This figure reflects two sources of inaccuracy in the lost cycles model. First, we have not accounted for two code fragments in our model — the initialization and finalization segments of the program, represented by nodes 2 and 4 in Figure 6.3. These code fragments grow in execution time with increasing n much more slowly than the rest of the program: hence for small n their effects are much more noticeable. The second source of inaccuracy concerns the resolution of our measurements. The measurements made by pp are accurate to microsecond resolution, but each profile is the sum of a number of such measurements, which can accumulate errors. The errors are only significant when overall running time is very small; however, Figure 6.6 shows that running



Execution Time (left) and Speedup (right) for $n = 1024$



Execution Time (left) and Speedup (right) for $n = 512$



Execution Time (left) and Speedup (right) for $n = 256$

Figure 6.5: Predicted and Actual Performance of TP 2D FFT for $n \geq 256$

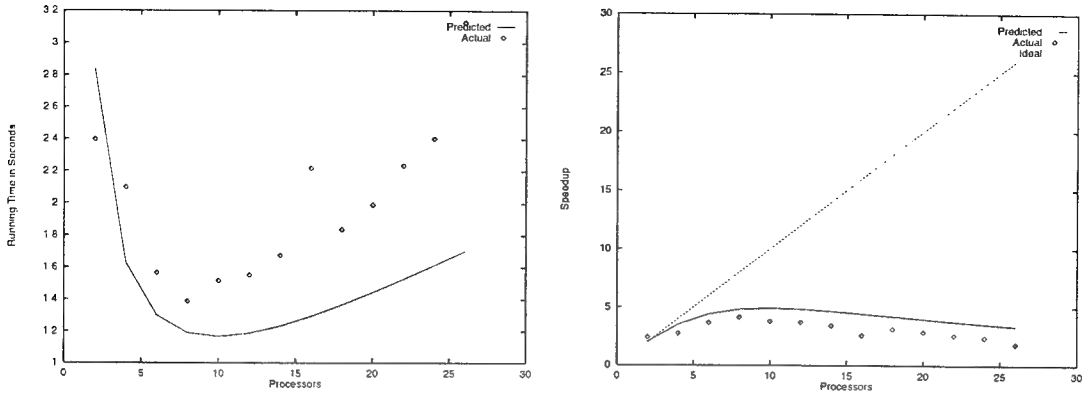


Figure 6.6: Predicted and Actual Performance of TP 2D FFT for $n = 64$

times are on the order of 1 - 3 seconds, small enough to be comparable to the accumulated errors in measurement.

This section shows that lost cycles modeling can accurately construct a performance model from the models of an application's code fragments. In the next section we will show how lost cycles modeling can predict the performance of an application in advance of implementation.

6.3.2 Predicting the Performance of DP 2D FFT

A primary goal of this thesis is to develop techniques for predicting the performance of applications in advance of implementation. The 2D FFT application provides an example we can use to demonstrate the utility of lost cycles modeling to achieve our goal. This section describes the construction of a lost cycles model for an (effectively) unimplemented version of 2D FFT.

The model developed in the last section was based on measurements of a particular implementation of 2D FFT - the task parallel implementation. From the same measurements used in the last section, we can construct a performance model for data parallel 2D FFT. This is done by simply reorganizing the source code into the data parallel form, and using lost cycles modeling to construct the performance model from the modified source.[1]

The reorganized main routine is shown in Figure 6.7. The new structure of the program consists of the four parallel loops executed in sequence (each subroutine contains a parallel loop as shown in Figure 6.2. Based on the new structure for the program, we can construct the parallel structure graph shown in Figure 6.8. The

[1]Eventually, we compiled and executed the data parallel version of 2D FFT, but only to gather performance data for comparison with model predictions.

```
c-------------------------------------------------------------
      program main
      real a(n,n,2,iters)
      call initialize_program
      do k=1,iters
         call initialize(a(k))
         call cffts(a(k))
         call transpose(a(k))
         call cffts(a(k))
      end do
      call print_performance
      end
c-------------------------------------------------------------
```

Figure 6.7: Edited Source for the Main Routine of Data Parallel 2D FFT

figure shows that nodes 3, 5, and 6 in the task parallel graph have been replaced by a single internal node in the data parallel graph — a serial node whose children are the four parallel loops.

The parallel structure shown in Figure 6.8 has only one internal node requiring processing (again overlooking nodes 2 and 4). Since it is a serial node, default rules specify that external variables remain unchanged when applied to component models. Thus each component model is supplied with n and p unchanged as inputs. Composition rules are similarly straightforward for this version. No structural rules apply, since this is a serial node, and no synchronization events are generated in this version. Modification is necessary to Communication Loss, since the transfer of the matrix between the initialization task and the transpose task does not occur in this version.

The results of performing lost cycles modeling on the new version of the application are shown in Figure 6.9 for $n \geq 512$. The figure shows that lost cycles modeling can accurately predict the performance of an application that has not been compiled or executed. Again, the model's accuracy is greatest for large values of n, which are shown in the figure.

The utility of lost cycles modeling can be seen from Figure 6.10. This figure shows the predicted and actual values for both implementations ($n = 1024$). The figure shows that over most of the range of processors considered, the two models provide a reasonably accurate assessment of the measured difference between the two implementations. The models show that over the entire range, the data parallel version outperforms the task parallel version, agreeing with the conclusion from Section 5.3.2. Whereas in Section 5.3.2 it was necessary to perform lost cycles analysis on the new application in order to reach that conclusion, using

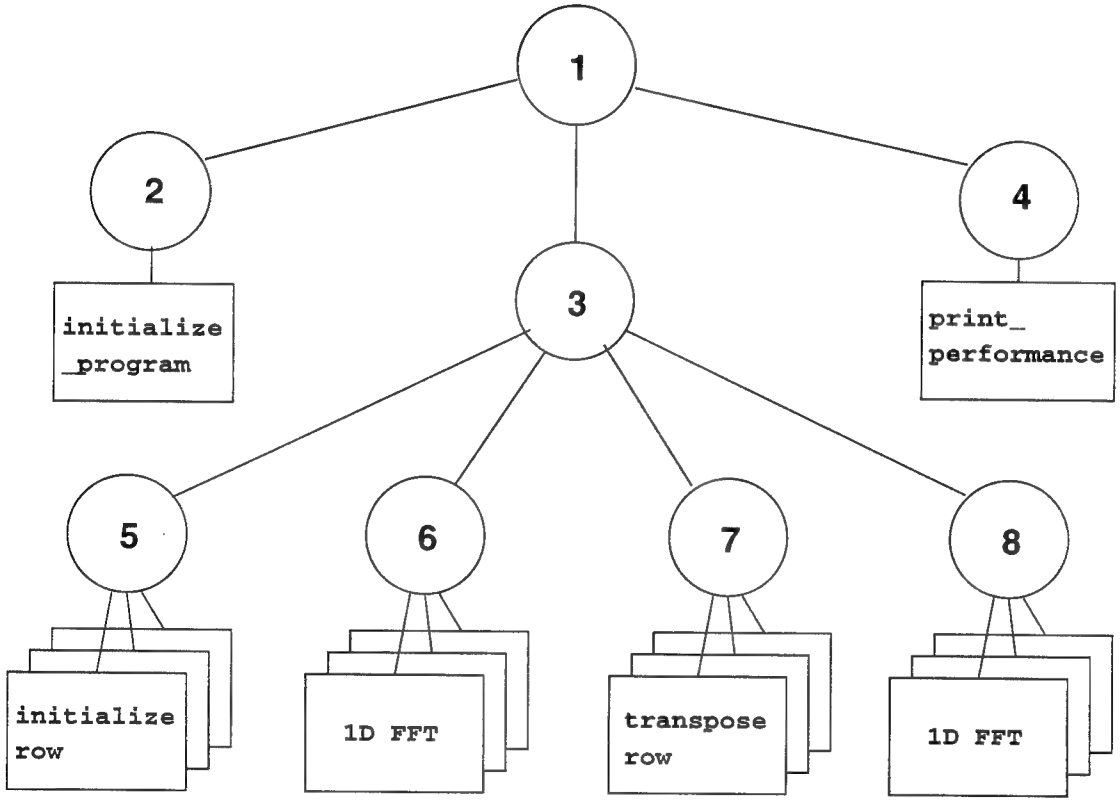
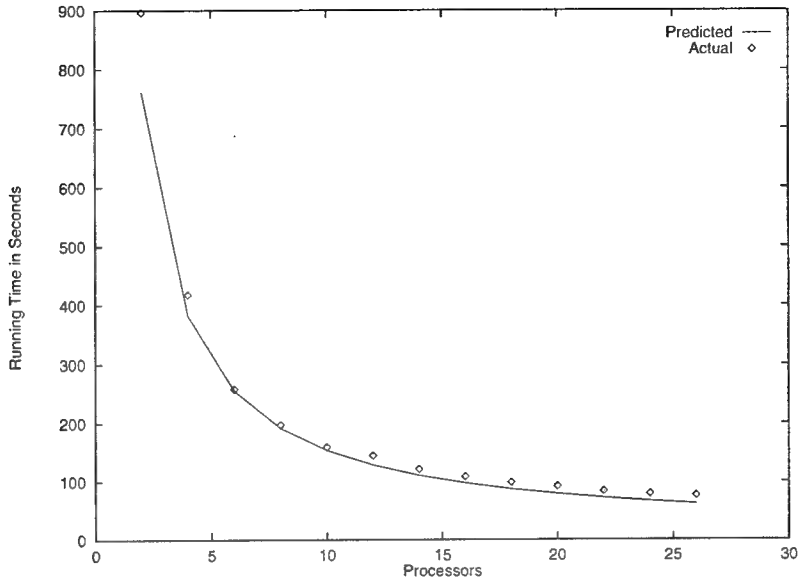
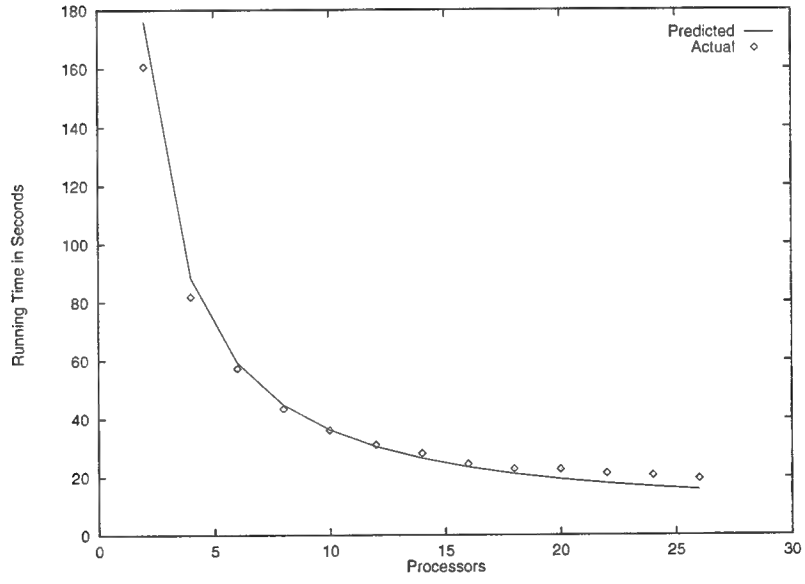


Figure 6.8: Parallel Structure for Data Parallel 2D FFT



Execution Time for $n = 1024$



Execution Time for $n = 512$

Figure 6.9: Predicted and Actual Execution Time for DP 2D FFT, $n \geq 512$

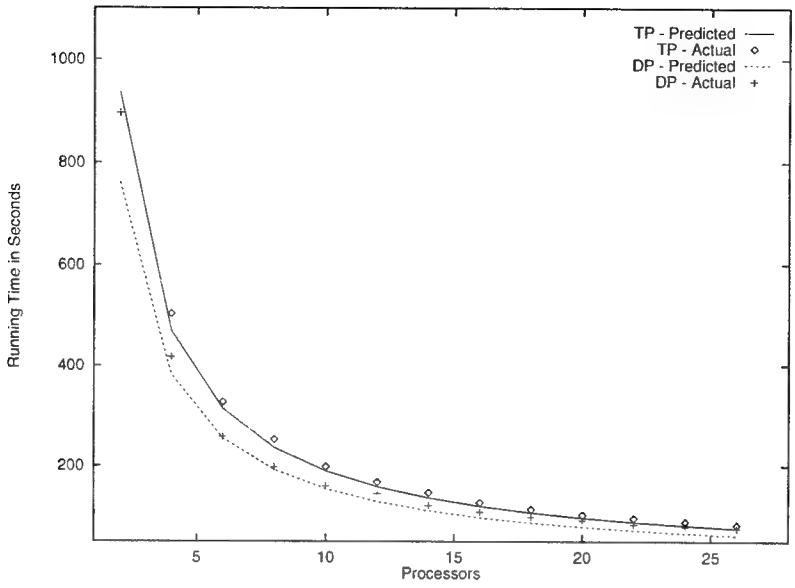


Figure 6.10: Lost Cycles Model Predictions for Two Implementations of 2D FFT

lost cycles modeling instead allows the same conclusion to be reached through a highly automated process.

With the completion of the `lcm` tool, our investigation of 2D FFT could be summarized as follows. After instrumenting the code for task parallel 2D FFT, lost cycles analysis is performed on each code fragment, resulting in a performance model for each fragment which is embedded in the source as Fortran comments. `lcm` is then run on the commented source, resulting in a performance model of the entire application. Figure 6.5 shows results obtained from such a model. Then, to explore the tradeoffs of an alternative parallelization, the program source is rearranged into the data parallel version, moving comments along with their corresponding code. `lcm` is then run on the new source, resulting in a new performance model. The results in Figure 6.9 show predictions based on the new model.

6.4 Summary

In this chapter we have shown a method for constructing an application's performance function $T_P(I, E)$ as a function both of execution-time variables (E) and the application's source code (I). We break the application up into fragments that are large enough to exhibit most of the significant performance effects in the application, but small enough to provide useful building blocks for application restructuring. Using lost cycles analysis, we create performance models for each

fragment; the performance models contain most of the important performance information for the application. Simple static analyses are then sufficient to compose the fragment models into an application model, and supply the remaining performance information necessary to create an accurate performance model of the application.

The benefit of lost cycles modeling derives from the flexibility of program rearrangement possible once the various fragment models have been built: any alternative parallel structure using those fragments can be evaluated automatically by `lcm`. Thus the programmer is given the freedom to experiment with alternative implementations without the need to fully implement them.

Our approach relies on dynamic measurement to solve the aspects of performance prediction that are difficult to solve statically, and uses static analysis to solve the aspects that are hard to solve empirically. In our approach, dynamic measurement provides machine and algorithmic constants, models of loop execution times, and models of communication costs. These are all requirements of performance prediction that are typically difficult to predict statically. On the other hand, our approach uses static analysis to evaluate the effects of code modification; this is appropriate since a measurement-based approach to evaluating alternative structures would require their full implementation and execution.

7 Conclusions

Parallel processing offers tremendous price/performance advantages, but reaping its benefits requires efficient use of machine resources. This disseration addresses the problem of finding efficient implementations for parallel applications, by presenting techniques and tools for performance measurement and prediction of parallel programs.

Our approach throughout this dissertation has been to decompose each problem in a structured way. Parallel overheads are decomposed into overhead categories that together meet the criteria of completeness, mutual exclusion, and meaning. These three criteria are necessary and sufficient for both performance tuning, and for performance prediction. The factors affecting parallel program performance are decomposed into internal and external effects, allowing a clear distinction between the role of dynamic and static techniques in our method of performance prediction. Program structure is decomposed into parallel structure and code fragments, supporting the maximum use of dynamic information to simplify static analysis.

7.1 Contributions

We have presented three techniques to facilitate the rapid development of efficient parallel programs: predicate profiling, lost cycles analysis, and lost cycles modeling.

Predicate profiling is a method of measuring the parallel overhead present in the execution of a parallel program. Predicate profiling works by assigning the measured overhead to categories (defined by *performance predicates*), which together meet three criteria: completeness, orthogonality, and meaningfulness. Completeness ensures that all the parallel overhead in an execution is captured; orthogonality ensures that no overhead is counted twice (thus overheads can be summed in a straightforward way); and meaningfulness ensures that overhead categories correspond to states of the execution that have significance in performance

tuning. Predicate profiling differs from previous measurement work in emphasizing these three criteria, and in its dual support for performance evaluation and performance prediction.

Section 2.3 pointed out that techniques for performance prediction of parallel applications include static and dynamic components. Our approach to performance prediction also contains dynamic and static components; the dynamic component, *lost cycles analysis,* is concerned exclusively with factors that are external to the source code. The static component of our approach, *lost cycles modeling,* is likewise restricted to factors internal to the source code.

Chapter 3 described external factors, which are the number of processors used to run the program, the size of the input data set, the structure of the input data set, the problem definition, and the machine used to run the program. Lost cycles analysis is a dynamic approach that empirically develops performance models of the application as a function of each of these external factors. Lost cycles analysis can address external factors using only runtime measurements because of the utility of overhead decomposition. Decomposing overheads simplifies the modeling process and exposes many typical behaviors that can be exploited to develop performance models.

Internal factors include the type of parallelism used and the choice of which code to parallelize. Lost cycles modeling takes a static approach to building performance models based on program structure, which is possible because of the leverage obtained from using lost cycles analysis in modelling code fragments. Since lost cycles analysis describes the performance of code fragments in terms of external factors, it captures the information necessary to describe how components of an application will behave when reorganized into a new parallel structure. This simplifies the static analysis required to construct the new application's performance model.

7.2 Future Directions

7.2.1 Performance Tuning

Predicate profiling is based on accurate measurements of machine-level events such as communication and contention. Unfortunately, this is not easy to do on many multiprocessors, which will limit the widespread use of predicate profiling. One solution is for hardware designers to design performance monitoring hardware into new processors; hopefully performance monitoring features like those found in the Alpha design [Digital Equipment Corporation, 1992] will become more widespread over time.

A useful extension of the predicate profiler would be the ability to provide procedure-level or line-level predicate profiles of an application. Highly detailed

profiles could be based on sampling the program counter as well as the performance predicates, and relating overheads to code segments at the source line level. This would address the problem presented by long-running applications with multiple distinct phases of execution, often with differing performance issues.

7.2.2 Performance Prediction

Performance prediction is not currently a tool that most parallel programmers use. While this thesis attempts to make performance prediction more tractable for parallel programmers, a number of questions need to be answered to fully solve the problems facing users.

The extent to which external factors in application performance are orthogonal is an issue in our approach. A high degree of interdependence of external factors could make lost cycles analysis inaccurate. The cases we have studied to date indicate that external factors are typically orthogonal, but more work needs to be done before we can be confident.

The best balance between static and dynamic analysis is still unclear in performance prediction. This thesis has attempted to use dynamic analysis to its fullest potential, but greater accuracy may be possible with a larger effort in static analysis.

A related question concerns the quantity of data that needs to be collected to accurately predict performance. The collection of large amounts of data for the purpose of performance prediction can in some cases require more effort than it saves. However, dynamic data provides answers to questions that are currently difficult or impossible to answer using static analysis. The best compromise between these goals is an open question in parallel performance prediction.

Bibliography

[Abrams *et al.*, 1992] Marc Abrams, Naganand Doraswamy, and Anup Mather, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains," *Journal of Parallel and Distributed Computing*, 3(6):672–685, November 1992.

[Aho *et al.*, 1988] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques and Tools*, Series in Computer Science. Addison Wesley, 1988.

[Amdahl, 1967] G. M. Amdahl, "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," In *AFIPS Conference Proceedings*, volume 20, pages 483–485. AFIPS Press, Reston, Va., April 1967.

[Anderson and Lazowska, 1990] Thomas E. Anderson and Edward D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance," In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, May 1990.

[Aral and Gertner, 1988] Ziya Aral and Ilya Gertner, "High-level Debugging in Parasight," In *Proceedings ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 151–162, May 1988.

[Atkinson and Donev, 1992] A. C. Atkinson and A. N. Donev, *Optimum Experimental Design*, Oxford Statistical Science Series. Oxford Science Publications, 1992.

[Bailey *et al.*, 1994] David H. Bailey, Eric Barszcz, Leonardo Dagum, and Horst D. Simon, "NAS Parallel Benchmark Results 4-94," Technical Report RNR-94-006, NASA Ames Research Center, 1994.

[Balasundaram *et al.*, 1991] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions," In *Proceedings of PPoPP 91*, pages 213–223. ACM Sigplan Notices, 1991.

[Ball and Larus, 1992] Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," In *Conference Record of the Nineteenth POPL*, Albuquerque, NM, 19-22 January 1992.

[Becker *et al.*, 1988] Richard A. Becker, John M. Chalmers, and Allan R. Wilks, *The New S Language: A Programming Environment for Data Analysis and Graphics*, Computer Science Series. Wadsworth & Brooks / Cole, Pacific Grove, California, 1988.

[Bodin *et al.*, 1990] F. Bodin, D. Windheiser, W. Jalby, D. Atapattu, M. Lee, and D. Gannon, "Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000," In *Proceedings of the 1990 International Conference on Supercomputing*, pages 401–413, Amsterdam, The Netherlands, June 1990.

[Box *et al.*, 1978] George E. P. Box, William G. Hunter, and J. Stuart Hunter, *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*, Wiley Series in Probability and Mathematical Statistics. John Wiley and Sons, Inc., 1978.

[Brochard and Freau, 1990] L. Brochard and A. Freau, "Designing Algorithms on Hierarchical Memory Multiprocessors," In *Proceedings of the 1990 ACM Conference on Supercomputing*, pages 414–427. ACM, 1990.

[Burkhart and Millen, 1989] Helmar Burkhart and Roland Millen, "Performance Measurement Tools in a Multiprocessor Environment," *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[Callahan *et al.*, 1990] David Callahan, Ken Kennedy, and Allan Porterfield, "Analyzing and Visualizing Performance of Memory Hierarchies," In *Performance Instrumentation and Visualization*, pages 1–26. ACM Press, 1990.

[Carmona and Rice, 1991] Edward A. Carmona and Michael D. Rice, "Modeling the Serial and Parallel Fractions of a Parallel Algorithm," *Journal of Parallel and Distributed Computing*, 13:286–298, 1991.

[Char *et al.*, 1991] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. W. Watt, *Maple V Language Reference Manual*, Springer-Verlag, New York, 1991.

[Cheriton *et al.*, 1991] David R. Cheriton, Hendrik A. Goosen, and Philip Machanick, "Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience," *Proceedings of the International Symposium on Shared-Memory Multiprocessing*, pages 109–118, 1991.

[Clement and Quinn, 1993] Mark J. Clement and Michael J. Quinn, "Analytical Performance Prediction on Multicomputers," In *Proceedings of Supercomputing '93*, pages 886–894, November 1993.

[Clement and Quinn, 1994] Mark J. Clement and Michael J. Quinn, "Symbolic Performance Prediction of Scalable Parallel Programs," *Unknown*, 1994.

[Crovella *et al.*, 1992] Mark Crovella, Ricardo Bianchini, Thomas LeBlanc, Evangelos Markatos, and Robert Wisniewski, "Using Communication-to-Computation Ratio in Parallel Program Design and Performance Prediction," In *Proceeedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, December 1992.

[Crowl *et al.*, 1994] Lawrence A. Crowl, Mark Crovella, Thomas J. LeBlanc, and Michael L. Scott, "The Advantages of Multiple Parallelizations in Combinatorial Search," *Journal of Parallel and Distributed Computing*, 21(1):110–123, April 1994.

[Culler *et al.*, 1993] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," In *Proceedings of the Fourth PPOPP*, San Diego, CA, 20-22 May 1993.

[Cybenko *et al.*, 1991] G. Cybenko, J. Bruner, S. Ho, and S. Sharma, "Parallel Computing and the Perfect Benchmarks," In *Intl. Symposium on Supercomputing*, Fukwoka, Japan, November 1991.

[Davis and Hennessy, 1988] Helen Davis and John Hennessy, "Characterizing the Synchronization Behavior of Parallel Programs," In *Proceedings of the First PPEALS*, pages 198–211, July 1988.

[Digital Equipment Corporation, 1992] Digital Equipment Corporation, "DEC-Chip 21064-AA RISC Microprocessor Preliminary Data Sheet," Digital Equipment Corporation, Maynard, MA, 1992.

[Dimpsey and Iyer, 1991] R. T. Dimpsey and R. K. Iyer, "Performance Prediction and Tuning on a Multiprocessor," In *Proceedings of the Eighteenth ISCA*, pages 190–199, Toronto, Canada, May 1991.

[Dongarra *et al.*, 1990] J. Dongarra, O. Brewer, J. A. Kohl, and S. Fineberg, "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *Journal of Parallel and Distributed Computing*, 9(2):185–202, June 1990.

[Eager and Zahorjan, 1993] Derek L. Eager and John Zahorjan, "Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing," *ACM Transactions on Computer Systems*, 11:1–32, February 1993.

[Eager *et al.*, 1989] Derek L. Eager, John Zahorjan, and Edward D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[Fahringer, 1994] Thomas Fahringer, "Evaluation of Benchmarking Performance Estimation for Parallel Fortran Programs on Massively Parallel SIMD and MIMD Computers," In *IEEE Proc. of the 2nd Euromicro Workshop on Parallel and Distributed Processing, Malaga, Spain*, January 1994.

[Fahringer and Zima, 1993] Thomas Fahringer and Hans P. Zima, "A Static Parameter Based Performance Prediction Tool for Parallel Programs," In *Proceedings of International Conference on Supercomputing*, pages 207–219. ACM SIGARCH, ACM Press, July 20-22 1993.

[Flatt and Kennedy, 1989] Horace P. Flatt and Ken Kennedy, "Performance of Parallel Processors," *Parallel Computing*, 12:1–20, 1989.

[Gallivan *et al.*, 1991] K. Gallivan, W. Jalby, A. Maloney, and H. Wijshoff, "Performance Prediction for Parallel Numerical Algorithms," *International J. of High Speed Computing*, 3(1):31–62, 1991.

[Goldberg and Hennessy, 1993] Aaron J. Goldberg and John L. Hennessy, "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[Graham *et al.*, 1982] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," In *SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, Boston, June 1982.

[Grama *et al.*, 1993] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar, "Isoefficiency function: A scalability metric for parallel algorithms and architectures," *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, 1993.

[Grama and Kumar, 1992] Anarh Y. Grama and Vipin Kumar, "Scalability Analysis of Partitioning Strategies for Finite Element Graphs: A Summary of Results," In *Proceedings Supercomputing '92*, pages 83–92, Minn., MN, November 1992. IEEE.

[Gumbel, 1954] E. J. Gumbel, "The Maxima of the Mean of the Largest Value of the Range," *Annals of Mathematical Statistics*, 25:76 -84, 1954.

[Gustafson, 1988] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, 31(5):532–533, May 1988.

[Gustafson *et al.*, 1988] J.L. Gustafson, G.R. Montry, and R.E. Benner, "Development of Parallel Methods for a 1024-processor Hypercube," *SIAM J. on SSTC*, 9(4), July 1988.

[Hartley and David, 1954] H. O. Hartley and H. A. David, "Universal Bounds for Mean Range and Extrema Observations," *Annals of Mathematical Statistics*, 25:85–99, 1954.

[Heath and Etheridge, 1991] Michael T. Heath and Jennifer A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, 8(5):29–39, September 1991.

[Hickey *et al.*, 1992] Timothy J. Hickey, Jacques Cohen, Hirofumi Hotta, and Thierry Petitjean, "Computer-Assisted Microanalysis of Parallel Programs," *ACM Transactions on Programming Languages and Systems*, 14(1):54–106, January 1992.

[Hollingsworth and Miller, 1993] Jeffrey K. Hollingsworth and Barton P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," In *7th ACM International Conference on Supercomputing*, July 1993.

[Hummel *et al.*, 1992] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Communications of the ACM*, 35(8):90–101, August 1992.

[Jain, 1991] Raj Jain, *The Art of Computer Systems Performance Analysis*, Wiley and Sons, Inc., 1991.

[Kaelbling and Ogle, 1990] Michael J. Kaelbling and David M. Ogle, "Minimizing Monitoring Costs: Choosing Between Tracing and Sampling," In *Proceedings of the 23rd Hawaii International Conference on Systems Sciences*, pages 314 – 320, January 1990.

[Karp and Flatt, 1990] A. H. Karp and H. P. Flatt, "Measuring Parallel Processor Performance," *Communications of the ACM*, 33(5):539–543, May 1990.

[Karp and Ramachandran, 1990] R. M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, 1990.

[Kendall Square Research, 1991] Kendall Square Research, "KSR1 Principles of Operation," Kendall Square Research, 170 Tracer Lane, Waltham MA, 15 October 1991.

[Kendall Square Research, 1992] Kendall Square Research, "KSR1 Fortran Programming," Kendall Square Research, 170 Tracer Lane, Waltham MA, 15 February 1992.

[Kilpatrick and Schwan, 1991] Carol Kilpatrick and Karsten Schwan, "ChaosMON — Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems," In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1991, Published in SIGPLAN Notices, Jan, 1992.

[Knuth, 1976] D. E. Knuth, "Big Omicron and Big Omega and Big Theta," *SIGACT News*, 8(2):18–24, 1976.

[Kohn and Williams, 1993] James Kohn and Winifred Williams, "ATExpert," *Journal of Parallel and Distributed Computing*, 14, May 1993.

[Kumar *et al.*, 1994] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Computing*, Benjamin Cummings Publishing Co., 1994.

[Kumar and Gupta, 1991] Vipin Kumar and Anshul Gupta, "Analyzing Scalability of Parallel Algorithms and Architectures," Technical report, TR-91-18, Computer Science Department, University of Minnesota, June 1991.

[Lai and Sahni, 1984] Ten-Hwang Lai and Sartaj Sahni, "Anomalies in Parallel Branch and Bound Search," *Communications of the ACM*, 27(6), June 1984.

[Lazowska *et al.*, 1984] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik, *Quantitative System Performance*, Prentice-Hall Inc., Englewood Cliffs, NJ 07632, 1984.

[LeBlanc, 1988] T. J. LeBlanc, "Problem Decomposition and Communication Tradeoffs in a Shared Memory Multiprocessor," In *Numerical Algorithms for Modern Parallel Computer Architectures*, IMA Volumes in Mathematics and Its Applications. Springer-Verlag, 1988.

[LeBlanc *et al.*, 1990] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, 9:203–217, June 1990.

[LeBlanc and Mellor-Crummey, 1987] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[Lehr *et al.*, 1989] Ted Lehr, Zary Segall, Dalibor Vrsalovic, Eddie Caplan, Alan Chung, and Charles Fineman, "Visualizing Performance Debugging," *IEEE Computer*, pages 38–51, October 1989.

[Lockyer, 1964] K. G. Lockyer, *Introduction to Critical Path Analysis*, Pitman Publishing Co., New York, N.Y., 1964.

[Markatos *et al.*, 1991] Evangelos Markatos, Mark Crovella, Prakash Das, Cezary Dubnicki, and Thomas LeBlanc, "The Effects of Multiprogramming on Barrier Synchronization," In *Proceeedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, December 1991.

[Martonosi *et al.*, 1992] Margaret Martonosi, Anoop Gupta, and Thomas Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[Mehra *et al.*, 1994] P. Mehra, M. Gower, and M. Bass, "Automated Modeling of Message-Passing Programs," In *Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 94)*, pages 187–192, Durham, NC, Jan. 1994. IEEE Computer Society Press.

[Miller and Choi, 1988] Barton P. Miller and Jong-Deok Choi, "A Mechanism for Efficient Debugging of Parallel Programs," In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–144, June 1988.

[Miller *et al.*, 1990] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[Møller-Nielsen and Staunstrup, 1987] Peter Møller-Nielsen and Jørgen Staunstrup, "Problem-heap: A Paradigm for Multiprocessor Algorithms," *Parallel Computing*, 4:64–74, 1987.

[Nanda *et al.*, 1991] Arun K. Nanda, Honda Shing, Ten-Hwan Tzen, and Lionel M. Ni, "Resource Contention in Shared-Memory Multiprocessors: A Parameterized Performance Degradation Model," *Journal of Parallel and Distributed Computing*, 12:313–328, 1991.

[Netzer and Miller, 1992] Robert H. B. Netzer and Barton P. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," In *Proceedings Supercomputing '92*, pages 502–511, Minn., MN, November 1992. IEEE.

[Nicol and Willard, 1988] David M. Nicol and Frank H. Willard, "Problem Size, Parallel Architectures, and Optimal Speedup," *Journal of Parallel and Distributed Computing*, 5:404-420, 1988.

[Nudd *et al.*, 1993] G. R. Nudd, E. Papaefstathiou, Y. Papay, T. J. Atherton, C. T. Clarke, D. J. Kerbyson, A. F. Stratton, R. Ziani, and M. J. Zemerly, "A Layered Approach to the Characterisation of Parallel Systems for Performance Prediction," In *Performance Evaluation of Parallel Systems (PEPS) '93*, pages 26–34, U. Warwick, U.K., 29-30 November 1993.

[Pease *et al.*, 1991] D. Pease, A. Ghafoor, I. Ahmad, D. Andrews, K. Foudil-Bey, T. Karpinski, M. Mikki, and M. Zerrouki, "PAWS: A Performance Evaluation Tool for Parallel Computing Systems," *IEEE Computer*, pages 18–29, January 1991.

[Press *et al.*, 1988] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1988.

[Rao and Kumar, 1989] V. Nageshwara Rao and Vipin Kumar, "Parallel Depth-First Search," *International Journal of Parallel Processing*, 16(6), 1989.

[Rinard *et al.*, 1992] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam, "Heterogeneous Parallel Programming in Jade," In *Proceedings Supercomputing '92*, pages 245–256, Minn., MN, November 1992. IEEE.

[Rothberg and Gupta, 1990] Edward Rothberg and Anoop Gupta, "Parallel ICCG on a Hierarchical Memory Multiprocessor — Addressing the Triangular Solve Bottleneck," Technical Report CSL-TR-90-449, Department of Computer Science, Stanford University, September 1990.

[Singh *et al.*, 1991] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson, "Scalability of Parallel Sorting on Mesh Multicomputers," *International Journal of Parallel Processing*, 20(2), 1991.

[Sivasubramaniam *et al.*, 1994] Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran, "An Approach to Scalability Study of Shared Memory Parallel Systems," In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[Smith, 1990] C. U. Smith, *Performance Engineering of Software Systems*, The SEI Series in Software Engineering. Addison-Wesley Publishing Co., 1990.

[Snyder, 1986] L. Snyder, "Type Architectures, Shared Memory, and the Corollary of Modest Potential," *Annual Review of Computer Science*, 1, 1986.

[So *et al.*, 1987] K. So, A.S. Bolmarcich, F. Darema, and V.A. Norton, "A Speedup Analyzer for Parallel Programs," In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 653-662, August 1987.

[Sreekantaswamy *et al.*, 1991] H.V. Sreekantaswamy, S. Chanson, and A. Wagner, "Performance Prediction Modelling of Multicomputers," Technical Report 91-27, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada V6T 1W5, November 1991.

[Subhlok *et al.*, 1993] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross, "Programming Task and Data Parallelism on a Multicomputer," In *Proceedings of the Fourth PPOPP*, San Diego, CA, 20-22 May 1993.

[Tsuei and Vernon, 1990] Thin-Fong Tsuei and Mary K. Vernon, "Diagnosing Parallel Program Speedup Limitations Using Resource Contention Models," In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I–185 – I–189. The Pennsylvania State University Press, August 1990.

[Valiant, 1990] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, 33(8):103–111, August 1990.

[van Gemund, 1993] Arjan J.C. van Gemund, "Performance Prediction of Parallel Processing Systems: The PAMELA Methodology," In *Proc. 7th ACM Int. Conf. on Supercomputing*, pages 318–327, Tokyo, Japan, July 1993.

[Vrsalovic *et al.*, 1984] Dalibor Vrsalovic, Daniel P. Siewiorek, Zary Z. Segal, and Edward F. Gehringer, "Performance Prediction for Multiprocessor Systems," In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 139–146, 1984.

[Vrsalovic *et al.*, 1988] Dalibor Vrsalovic, Daniel P. Siewiorek, Zary Z. Segal, and Edward F. Gehringer, "Performance Prediction and Calibration for a Class of Multiprocessor Systems," *IEEE Transactions on Computers*, 37(11):1353–1365, November 1988.

[Yang and Miller, 1988] Cui-Qing Yang and Barton P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs," In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 366–373, 1988.

[Zhang and Srinivasan, 1990] X. Zhang and P. Srinivasan, "Distributed Task Processing Performance on a NUMA Shared Memory Multiprocessor," In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 786–789, December 1990.

[Zimran *et al.*, 1990] Eyal Zimran, Manohar Rao, and Zary Segall, "Performance Efficient Mapping of Applications to Parallel and Distributed Architectures," In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–147 – II–154, August 1990.

A Manual Pages

This appendix presents the user interface specifications for the tools pp and lca, respectively described in Section 4.2.2 and Section 5.1.2. The specifications are given in UNIX™ man(1) format.

A.1 Manual Page for pp

```
PP(1)                    UNIX Programmer's Manual                    PP(1)


NAME
     PP

SYNOPSIS
     pp [-f filename] [-d] [-c] [-i] [-E] [-B] [-m] [-t tag] [-l
     [lca_note]] [filename]

DESCRIPTION
     pp is a profiler based on work described in the paper, "Per-
     formance Debugging Using Parallel Performance Predicates" by
     M. Crovella and T. LeBlanc, in Proceedings of the ACM/ONR
     Workshop on Parallel and Distributed Debugging, May 1993.
     This is an experimental version but is being provided to
     Theory Center users to help determine where bottlenecks in
     their application programs might be.

     PP
      reads in the specified event log and breaks down the total
     time of the profiled program into 5 different categories:
```

Load Imbalance, Insufficient Parallellism, Synchronization Loss, Communication Loss, and Resource Contention. The data shown is total time for that category over all threads in the team.

To obtain the event log file that pp needs for input, one must instrument one's source program with calls to "pp" routines (see manpage for pplib), and set the PL_ELOG environment variable to 1 before running the program.

OPTIONS

-f <logfile> The name of the logfile (default: elog.log).
-d Present detailed debugging output. Useful for inspecting gist log files directly.
-c Present ceu_stall_time data (printing it out as "Cache Loss"). This data is not considered accurate in some circumstances.
-i Add the cost of inserted instructions into "Communication Loss".
-E The logfile is in ascii E format, as converted by gist.
-B The logfile is in ascii brief format, as converted by gist.
-m Multiple teams were created in the application. When using multiple teams, additional instrumentation must be added to the application. See pplib(3f).
-t <tag> The value of tag is used to delineate the code segments to be profiled in this run of pp. See pplib(3f) for more details.
-l [lca_note] The output of pp is concisely formatted in a form suitable as input to the tool lca(1). The optional string lca_note will be included on the output line; it can be used to identify the run for lca, and should take the form "<variable>=<value>".
filename - Name of the logfile. If omitted, the default is elog.log. You may also use redirection to specify the input file.

EXAMPLES

```
f77 -o runfast -r8 -O2 -para myprog.f -lctc -lpmon
```

```
        setenv PL_ELOG 1
        setenv PL_NUM_THREADS 5
        runfast
        PP

    Typical Output:

        PP version 4.0
        ** processors: 5
        Load Imbalance              0.180677
        Insuff Parallellism        16.813304
        Synchronization Loss        0.003779
        Communication Loss          2.274899
        Resource Contention         1.351362
        Total Time                 49.293890
        Remaining Time             28.669868


FILES
    pplib.o

SEE ALSO
    pplib(3f), gist

AUTHOR
    Mark Crovella, U. of Rochester Computer Science Dept.
```

A.2 Manual Page for pplib

```
pplib(3F)          UNIX Programmer's Manual          pplib(3F)


NAME
     pp_start_profiling, pp_create_team, pp_loop_start,
     pp_loop_end, pp_psect_start, pp_psect_end, pp_stop_team,
     pp_end_profiling, pp_synch_start, pp_synch_end

SYNOPSIS
     subroutine pp_start_profiling
     subroutine pp_end_profiling

     subroutine pp_start_profiling_tag(itag)
     subroutine pp_end_profiling_tag(itag)

     subroutine pp_create_team(iteam_id, num_threads)
     pp_stop_team(iteam_id)

     pp_loop_start(iteam_id)
     pp_loop_end(iteam_id)

     pp_psect_start(team_id)
     pp_psect_end(team_id)

     pp_synch_start
     pp_synch_end

     integer iteam_id, num_threads


DESCRIPTION
     These entry points are used to instrument your program to
     determine parallel overhead.  (See pp (1).) Profiling is
     started by a call to pp_start_profiling and completed by a
     call to pp_end_profiling. These calls may be placed anywhere
     in the program; they may appear multiple times in the pro-
     gram; and they may each be executed multiple times during an
     execution.  However, the thread that calls
     pp_start_profiling must be the thread that makes the next
```

call to pp_end_profiling, and these two calls must strictly alternate during execution.

Programs to be profiled must manage teams explicitly (many programs will only use a single team). Each team used in the program must be created using pp_create_team. pp_create_team creates a team of num_threads threads and returns the team ID in iteam_id. If num_threads is zero, pp_create_team uses the value of the environment variable PL_NUM_THREADS. If a team is to be destroyed explicitly, the program must call pl_stop_team first. Once created, teams must be specified in each parallel construct of the profiled section. For example,

```
c*ksr* user tile (i,teamid=iteam_id)

c*ksr* parallel sections (teamid=iteam_id)

c*ksr* parallel region (teamid=iteam_id)
```

In addition, synchronization operations must be bracketed with calls to pp_synch routines:

```
      subroutine wait(ivar)
      volatile ivar
      if (ivar .ne. 1) then
         call pp_synch_start
100      if (ivar .ne. 1) goto 100
         call pp_synch_end
      end if
      end
```

Finally, if more than one team is used in the application, then each parallel construct must be bracketed with calls to the appropriate pplib routines. For example:

```
         call pp_loop_start(iteam_id)
c*ksr* user tile (i,teamid=iteam_id)
         do 100 i=1,niters
         ...
   100 continue
```

```
c*ksr*endtile
      call pp_loop_end(iteam_id)
```

In this case, when using pp to generate the profile, it should be given the -m option.

Recompile the program as follows:

```
f77 -para -o pgm pgm.f -lctc -lpmon
```

Before running the program, set the environment variable PL_ELOG to 1.

As of version 4.3, pplib and pp now support the notion of profiling tags. By using the calls pp_start_profiling_tag and end_profiling_tag the programmer can independently and simultaneously profile multiple regions of code. The code regions profiled under different tags can overlap in any manner. These calls are used in exactly the same way as pp_start_profiling/pp_end_profiling, except that a user-supplied tag (integer) is supplied to the calls. This tag is then provided to pp, via the -t option, and pp reports only on code segments associated with that tag.

NOTES

You may have to put iteam_id into a common (or pass it as a parameter) if the parallel construct occurs within a subroutine.

EXAMPLE

Below, "myprof.f" uses a team of 5 threads for each parallel construct between calls to pp_start_profiling and pp_end_profiling The program is recompiled and executed to produce a trace file ("elog.out") that is then processed using pp.

```
f77 -o runfast -r8 -O2 -para myprof.f pplib.o -lpmon
setenv PL_ELOG 1
setenv PL_NUM_THREADS 5
runfast
pp
```

FILES

pplib.o

SEE ALSO

pp(1), gist(1)

AUTHOR

Mark Crovella, U. of Rochester Computer Science Department

A.3 Manual Page for lca

LCA(1) UNIX Programmer's Manual LCA(1)

NAME

LCA

SYNOPSIS

lca -c <category> -v <variable> -f <filename> [-d]
[-x <extraction-expression>] [-g <gnuplot-filename>]
[<optional-formulae>]

DESCRIPTION

lca is a tool for fitting performance data to analytic models, based on work described in the paper, "Parallel Performance Prediction Using Lost Cycles Analysis", by M. Crovella and T. LeBlanc, in Proceedings of Supercomputing '94.

lca reads in the specified datafile of performance data, and: 1) extracts the overhead data for <category>, additionally restricting it by any <extraction-expression>, 2) selects models that are appropriate to the particular overhead category, and 3) presents the goodness-of-fit for each of the models as a function of <variable>.

OPTIONS

-c <category> The overhead category of interest. Currently defined categories are ip (Insufficient Parallelism), li (Load Imbalance), sl (Syncrhonization Loss), cl (Communication Loss), and rc (Resource Contention). Additionally, lca can be used to model rt (Remaining Time - the pure computation in an execution) although it has no default models for rt.

-v <variable> The variable(s) under study. Typically these are p (number of processors) and d (data size). Multiple variables can be simultaneously fit by listing them separated by spaces, e.g., "d p". No default multi-variable models exist, but commandline-supplied models

can be used which specify variables as as x0, x1, etc (numbering in the order the variables are listed).
-f <datafile> The datafile contained the measured performance data, typically output from pp(1) using that program's -l flag.
-d Present considerable amounts of debug output.
-x <extraction-expression> Normally lca will extract all the records from <datafile> that refer to both <category> and <variable>. However, in cases where multiple variables are being studied, only a subset of those records may be of interest. For example, from data varying both p and d, one might only be interested in fitting a model to those performance records where, say, p=32. This would then be specified as such after the -x option.
-g For single-variable fits, create gunplot files that can be used to graphically inspect the quality of each fit ("Chi-by-eye"). These files are named <gnuplot-filename>.data and <gnuplot-filename>.gp. Starting gnuplot and executing a "load <gnuplot-filename>" will then present the plot.
<optional-formulae> In many cases the user may wish to add additional formulae to those selected by lca based on the category. Any additional formulae present on the command-line (specified in terms of x0, x1, etc) will be processed by lca. Multiple formulae should be separated by spaces and the entire set should be surrounded by quotes.

SEE ALSO

pp(1), pplib(3f)

AUTHOR

Mark Crovella, U. of Rochester Computer Science Dept.